30th European Conference on Object-Oriented Programming

ECOOP'16, July 18–22, 2016, Rome, Italy

Edited by

Shriram Krishnamurthi

Benjamin S. Lerner

Editors

Shriram Krishnamurthi
Brown University
Providence, RI, USA
sk@cs.brown.edu

Benjamin S. Lerner
Northeastern University
Boston, MA, USA
blerner@ccs.neu.edu

ACM Classification 1998
D.1 Programming Techniques and D.2 Software Engineering

ISBN 978-3-95977-014-9

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-014-9.

Publication date
July 2016

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/LIPIcs.ECOOP.2016.0

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

30th European Conference on Object-Oriented Programming (ECOOP 2016).
Editors: Shriram Krishnamurthi and Benjamin S. Lerner
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

ECOOP continues to evolve. It has come a long way form its "object-oriented" roots (and it has even, once, traveled a long way from its European roots). It is difficult to pin down exactly what belongs in ECOOP, yet I believe many attendees feel there is something distinctive (and valuable) about the conference.

Part of its distinction is in the format: not just that it is in Europe in the summer, but that it continues to hew to a single track with a room of informed and attentive listeners, an experience that has largely vanished in the rest of the programming and programming languages world. But I believe there is also something identifiable about its content.

Call for Papers

In preparing this year's Call for Papers, I did a small analysis of the calls from the past several years. The result was (at least to me) amusing, and can be seen here:

`https://github.com/shriram/ecoop-cfps`

Program chairs, it appears, cannot help but chop and change the call, with most of their attention devoted to the laundry list of topics. Well, I could not resist it either, but I also believe our conference calls would be greatly improved by removing these laundry lists and replacing them with reasonably crisp, non-vacuous, non-trivial statements of purpose. To that end, I coined the following:

> [ECOOP's] primary focus has been object-orientation, though it is liberal in its taste and, in recent years, has accepted quality papers over a much broader range of programming language and programming topics. Its sweet spot tends to be the theory, design, implementation, optimization, and analysis of programming languages that enable or enforce abstractions—data, control, security, performance—across various programming styles, from object-orientation to reactivity to spreadsheets. It encourages both innovative and creative solutions to real problems, and evaluations of existing solutions in ways that shed new insights. Following recent precedent, it also encourages the submission of reproduction studies.

I look forward to seeing what future chairs will make of it.

Submission Date

This year, we also moved ECOOP's submission deadline earlier. ECOOP has long set its date in reaction to when ESOP's notifications—rejections, specifically—come out. But I felt having a deadline just a few days later suggests that the feedback from ESOP can in fact be attended to that quickly, which quite often is false—thereby disrespecting the hard work of the ESOP reviewers, and pushing the community towards a "dart-throwing" approach to paper submissions. Waiting for ESOP also pushed a non-trivial load of work to the winter break period at the end of the year (around Christmas and other festivities), which felt improper to impose on PC members who are, after all, volunteers.

Therefore, ECOOP had its deadline before ESOP's responses. This almost certainly contributed to a significant drop in submissions—we had only 79—but interestingly, the size of the final accepted paper list was not really smaller than usual, nor was there any perception of a drop in quality. Perhaps not waiting for rejected papers is not such a problem

30th European Conference on Object-Oriented Programming (ECOOP 2016).
Editors: Shriram Krishnamurthi and Benjamin S. Lerner
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

after all, except inasmuch as it fails to boost the denominator of the acceptance rate, affecting "prestige".

Reviewing

Thanks to the smaller number of submissions, we were able to lavish attention on the papers. No paper (with the exceptions below) received fewer than three reviews, of course, but *many* papers received five or even six. Because nobody had more than ten papers to review, the reviewers put a very serious effort into it, and produced outstanding work. Just about every reviewer produced a useful set of questions for authors to answer and then took their responses seriously. A later survey indicated significant satisfaction from authors, even of rejected papers. I doubt papers have been as thoroughly reviewed for any conference in recent years.

We tried a new-ish format that was only partially successful. The idea was that, in the two weeks after the submission deadline (finishing before Christmas), we would run a "kick the tires" round (inspired by what Blackburn and Hauswirth put in place for artifacts). Each paper would get two PC reviewers, and their job was to:

- Check for anonymity failures.
- Identify other reviewers.
- Identify potential conflicts of interest.
- Confirm that the paper was a submission worthy of full reviewing.
- Determine if their interest was reduced now that they had seen the paper, so we could find other reviewers.
- Inform the chair if they spotted any other issues.

Some of these goals were indeed achieved. Multiple reviewers came across papers that were not what they had expected from the title and abstract, and were hence significantly less interested in. Sometimes these problems are only discovered during the very last few days of the review period; this made sure that such problems were (partially) detected in the *first* few days instead, and reviewers were re-allocated elsewhere.

Other goals did not succeed as much. A few anonymization failures were discovered early, and we were able to get fixed versions of papers before regular reviewing began. But several were missed during this round and only uncovered later.

Finally, for non-viable papers, once there was unanimous agreement between the two assigned reviews and me, in lieu of regular reviews I wrote a generic chair's response. This enabled reviewers to focus their energy on papers worth their attention. Both at this ECOOP and at last year's Onward!, where I did something similar, this somewhat backfired. Sometimes the authors of papers that are rejected early are the least aware of how reviewing works, and are liable to get upset and badger chairs about due process, the nature of science, and so on. Having done this twice, I've learned a lesson: let the papers just get three short reviews and don't try to save time. In the long run, it costs more time than it saves.

PC Meeting

In keeping with ECOOP tradition, we had a live PC meeting. I was not initially a big fan of the idea: I'm not sure how much live meetings accomplish, and they also seem to impose big costs. Therefore, I asked on social media, and was surprised by the strong support for a live meeting (and vociferous opposition to doing things electronically). I also polled my potential PC members, and they too overwhelmingly voted in favor of a live meeting.

The meeting took one day, and was followed by a 2.5 day workshop featuring PC speakers. I hosted the attendees at Brown, which is just outside Boston, MA, so the PC workshop had attendees from the greater Boston area attend. I believe the meeting and workshop were a success.

Outcome

One thing the PC quickly agreed on is that authors have a tendency to inflate claims about their work. I believe this is a natural consequence of the punishing standards that PCs set; authors therefore feel it essential to present their results as acts of perfection, rather than as they actually are: works in progress. But this complicates the paper record, and forces later authors to compare against what might be exaggerated or even irrelevant claims.

We therefore agreed that we would be liberal in shepherding papers. This is not common in programming languages conferences, and can only work with PC members who are willing to commit to doing the extra work after the date they thought their service would be over; happily, this PC was very willing to perform this extra work. (On the other hand, in some communities it is standard for *all* papers to be assigned a shepherd.) Therefore, we accepted only ten papers outright (with the usual expectation that authors will make some revisions, but not a checked, formal requirement that they do so). In contrast, sixteen papers were sent to shepherding.

Of the sixteen, fifteen passed muster, most of them quite quickly and easily. One paper had repeated problems, and I decided the PC had already put in too much effort without a clear end in sight. Therefore, that paper was rejected, leaving us with 25 accepted out of 79. (Incidentally, that one rejected paper also had non-trivial problems found by the Artifact Evaluation Committee. It is unclear what would have happened if our shepherd had not caught the issues the artifact raised.)

Conclusion

It is always healthy to ask whether our ongoing activities are viable and worthwhile. I'm happy to say that I believe ECOOP is. It provides a distinctive, high-quality venue, and the surrounding constellation of activities—such as the summer school, CurryOn, workshops, etc.—have given it a strong supporting context. The conference is healthy and lively.

Many thanks to my supporting team at Brown, who ran the PC meeting and workshop: Jack Wrenn, Justin Pombrio, Hannah Quay-de la Vallee, Tim Nelson, Kate Correia, and Lauren Clarke. Ben Lerner has handled assembling the proceedings with saintly patience. Jan Vitek was an endless source of gnomic wisdom. Camil Demetrescu, Emilio Coppa, and Daniele Cono D'Elia have made various things that should have been hard startlingly easy. Beppe Castagna, Richard Jones, and Sophia Drossopoulou fielded numerous questions as prior chairs, providing responses not only wise but also laced with wit. Finally, the AITO SC was encouragingly positive about various proposals. A conference that can attract this many accomplished people to run it with passion has a bright future indeed.

Shriram Krishnamurthi
Providence, RI, USA
2016-05-29

List of Authors

Jonathan Aldrich
Carnegie Mellon University
Pittsburg, PA, USA
jonathan.aldrich@cs.cmu.edu

Karim Ali
TU Darmstadt
Darmstadt, Germany
karim.ali@cased.de

Esben Andreasen
Aarhus University
Aarhus, Denmark
esbena@cs.au.dk

Pavel Avgustinov
Semmle
Oxford, England
pavel@semmle.com

Edd Barrett
King's College London
London, England
edward.barrett@kcl.ac.uk

Eric Bodden
Paderborn University & Fraunhofer IEM
Paderborn, Germany
eric.bodden@uni-paderborn.de

Carl Friedrich Bolz
King's College London
London, England
cfbolz@gmx.de

Kim Bruce
Pomona College
Pomona, CA, USA
kim@cs.pomona.edu

Luis Caires
Universidade Nova de Lisboa
Lisbon, Spain
lcaires@fct.unl.pt

Elias Castegren
Uppsala University
Uppsala, Sweden
elias.castegren@it.uu.se

Satish Chandra
Samsung Research America
Mountain View, CA, USA
schandra@samsung.com

Guoyang Chen
North Carolina State University
Raleigh, NC, USA
gychen1991@gmail.com

Sheng Chen
UL Lafayette
Lafayette, LA, USA
chen@louisiana.edu

Maxime Chevalier-Boisvert
Université de Montréal
Montréal, Canada
maximechevalierb@gmail.com

Joeri De Koster
Vrije Universiteit Brussel
Brussels, Belgium
jdekoste@vub.ac.be

Wolfgang De Meuter
Vrije Universiteit Brussel
Brussels, Belgium
wdmeuter@vub.ac.be

Lukas Diekmann
King's College London
London, England
lukas.diekmann@gmail.com

Jens Dietrich
Massey University
Palmerston North, New Zealand
j.b.dietrich@massey.ac.nz

Martin Erwig
Oregon State University
Corvallis, OR, USA
erwig@oregonstate.edu

Jonathan Eyolfson
University of Waterloo
Waterloo, Ontario, Canada
jeyolfso@uwaterloo.ca

30th European Conference on Object-Oriented Programming (ECOOP 2016).
Editors: Shriram Krishnamurthi and Benjamin S. Lerner
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Marc Feeley
Université de Montréal
Montréal, Canada
feeley@iro.umontreal.ca

Luminous Fennell
University of Freiburg Freiburg, Germany
fennell@informatik.uni-freiburg.de

Yossi Gil
Technion — Israel Institute of Technology
Haifa, Israel
yogi@cs.technion.ac.il

Colin S. Gordon
Drexel University
Philadelphia, PA, USA
csgordon@cs.drexel.edu

Danny M. Groenewegen
Delft University of Technology
Delft, Netherlands
d.m.groenewegen@tudelft.nl

Dan Grossman
University of Washington
Seattle, WA, USA
djg@cs.washington.edu

Daco C. Harkes
Delft University of Technology
Delft, Netherlands
d.c.harkes@tudelft.nl

Michael Homer
Victoria University of Wellington
Wellington, New Zealand
mwh@ecs.vuw.ac.nz

Zhenjiang Hu
National Institute of Informatics
Tokyo, Japan
hu@nii.ac.jp

Kamil Jezek
University of West Bohemia
Pilsen, Czech Republic
kjezek@kiv.zcu.cz

Jiajun Jiang
Peking University
Beijing, China
xgdsmileboy@gmail.com

Michael Peyton Jones
Semmle
Oxford, England
michael@semmle.com

Timothy Jones
Victoria University of Wellington
Wellington, New Zealand
tim@ecs.vuw.ac.nz

Jaakko Järvi
Texas A&M University
College Station, TX, USA
jarvi@cse.tamu.edu

Daisuke Kikuchi
Tohoku University and Hitachi Solutions
East Japan, Ltd.
Sendai, Japan
kikuchi@riec.tohoku.ac.jp

Darya Kurilova
Carnegie Mellon University
Pittsburg, PA, USA
darya@cs.cmu.edu

Patrick Lam
University of Waterloo
Waterloo, Ontario, Canada
patrick.lam@uwaterloo.ca

Tomer Levy
Technion — Israel Institute of Technology
Haifa, Israel
stlevy@campus.technion.ac.il

Jun Li
Peking University
Beijing, China
j.lee.pku23@gmail.com

Yue Li
UNSW Australia
Sydney, Australia
yueli@cse.unsw.edu.au

Chunhua Liao
Lawrence Livermore National Laboratory
Livermore, CA, USA
liao6@llnl.gov

Xiangyu Luo
Peking University
Beijing, China
vani@pku.edu.cn

Filipe Militao
Carnegie Mellon University & Universidade Nova de Lisboa
Pittsburg, PA, USA
fm2097ad+ecoop2016@gmail.com

Lisa Nguyen Quang Do
Fraunhofer IEM
Paderborn, Germany
lisa.nguyen@sit.fraunhofer.de

James Noble
Victoria University of Wellington
Wellington, New Zealand
kjx@ecs.vuw.ac.nz

Pierre Néron
French Network and Information Security Agency (ANSSI)
Paris, France
pierre.neron@ssi.gouv.fr

Atsushi Ohori
Tohoku University
Sendai, Japan
ohori@riec.tohoku.ac.jp

Zachary Palmer
Swarthmore College
Swarthmore, PA, USA
zachary.palmer@swarthmore.edu

Sean Parent
Adobe Systems, Inc.
San Josa, CA, USA
sparent@adobe.com

Casper Bach Poulsen
Delft University of Technology
Delft, Netherlands
c.b.poulsen@tudelft.nl

Tomohiro Sasaki
Tohoku University
Sendai, Japan
tsasaki@riec.tohoku.ac.jp

Alceste Scalas
Imperial College London
London, England
alceste.scalas@imperial.ac.uk

Max Schaefer
Semmle
Oxford, UK
max@semmle.com

Koushik Sen
UC Berkeley
Berkeley, CA, USA
ksen@berkeley.edu

Xipeng Shen
North Carolina State University
Raleigh, NC, USA
xshen5@ncsu.edu

Scott F. Smith
Johns Hopkins University
Baltimore, MD, USA
scott@cs.jhu.edu

Johannes Späth
Fraunhofer SIT
Paderborn, Germany
johannes.spaeth@sit.fraunhofer.de

Manu Sridharan
Samsung Research America
Mountain View, CA, USA
manu@sridharan.net

Janwillem Swalens
Vrije Universiteit Brussel
Brussels, Belgium
jswalens@vub.ac.be

Tian Tan
UNSW Australia
Sydney, Australia
tiantan@cse.unsw.edu.au

Peter Thiemann
University of Freiburg
Freiburg, Germany
thiemann@informatik.uni-freiburg.de

Frank Tip
Samsung Research America
Mountain View, CA, USA
ftip@samsung.com

Andrew Tolmach
Portland State University
Portland, OR, USA
tolmach@pdx.edu

John Toman
University of Washington
Seattle, WA, USA
jtoman@cs.washington.edu

Laurence Tratt
King's College London
London, England
laurie@tratt.net

Katsuhiro Ueno
Tohoku University
Sendai, Japan
katsu@riec.tohoku.ac.jp

Eelco Visser
Delft University of Technology
Delft, Netherlands
visser@acm.org

Eric Walkingshaw
Oregon State University
Corvallis, OR, USA
walkiner@oregonstate.edu

Chenglong Wang
Peking University
Beijing, China
clwang@cs.washington.edu

Tobias Wrigstad
Uppsala University
Uppsala, Sweden
tobias.wrigstad@it.uu.se

Yingfei Xiong
Peking University
Beijing, China
xiong.yingfei@gmail.com

Jingling Xue
UNSW Australia
Sydney, Australia
jingling@cse.unsw.edu.au

Nobuko Yoshida
Imperial College London
London, England
n.yoshida@imperial.ac.uk

Lu Zhang
Peking University
Beijing, China
zhanglucs@pku.edu.cn

Yifei Zhang
UNSW Australia
Sydney, Australia
yzhang@cse.unsw.edu.au

Yue Zhao
North Carolina State University
Raleigh, NC, USA
yzhao30@ncsu.edu

Oege de Moor
Semmle
Oxford, UK
oege@semmle.com

Trace Typing: An Approach for Evaluating Retrofitted Type Systems

Esben Andreasen[1], Colin S. Gordon[2], Satish Chandra[3], Manu Sridharan[4], Frank Tip[5], and Koushik Sen[6]

1 **Aarhus University**
esbena@cs.au.dk
2 **Drexel University**
csgordon@cs.drexel.edu
3 **Samsung Research America**
schandra@samsung.com
4 **Samsung Research America**
m.sridharan@samsung.com
5 **Samsung Research America**
ftip@samsung.com
6 **UC Berkeley**
ksen@berkeley.edu

Abstract

Recent years have seen growing interest in the retrofitting of type systems onto dynamically-typed programming languages, in order to improve type safety, programmer productivity, or performance. In such cases, type system developers must strike a delicate balance between disallowing certain coding patterns to keep the type system simple, or including them at the expense of additional complexity and effort. Thus far, the process for designing retrofitted type systems has been largely ad hoc, because evaluating multiple variations of a type system on large bodies of existing code is a significant undertaking.

We present *trace typing*: a framework for automatically and *quantitatively* evaluating variations of a retrofitted type system on large code bases. The trace typing approach involves gathering traces of program executions, inferring types for instances of variables and expressions occurring in a trace, and merging types according to merge strategies that reflect specific (combinations of) choices in the source-level type system design space.

We evaluated trace typing through several experiments. We compared several variations of type systems retrofitted onto JavaScript, measuring the number of program locations with type errors in each case on a suite of over fifty thousand lines of JavaScript code. We also used trace typing to validate and guide the design of a new retrofitted type system that enforces fixed object layout for JavaScript objects. Finally, we leveraged the types computed by trace typing to automatically identify tag tests — dynamic checks that refine a type — and examined the variety of tests identified.

1998 ACM Subject Classification F.3.3 Studies of Program Constructs

Keywords and phrases Retrofitted type systems, Type system design, trace typing

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.1

1 Introduction

In recent years, there have been a number of efforts to *retrofit* [21] type systems onto dynamically-typed languages, to aid developer productivity, correctness, and performance.

These languages are of increasing importance, primarily due to their common use in web applications on both the client and server side. As more large-scale, complex programs are written in such languages, greater need arises for static types, due to the resulting benefits for static error checking, developer tools, and performance. Recent high-profile projects in type system retrofitting include Closure [16], TypeScript [5] and Flow for JavaScript [13], and Hack for PHP [32]. Given the proliferation of dynamically-typed languages, there are many retrofitted type systems ahead.

The key questions in the design of these type systems involve selecting which features to include. Richer features may allow more coding patterns to be understood and validated by the type checker. However, richer type systems come at the cost of greater complexity and implementation effort, and it is rarely a priori obvious whether a given feature's benefit outweighs its implementation burden. Hence, great care must be taken in deciding if a particular type system feature is worthy of inclusion. Such decision points continue to arise as a type system evolves: e.g., TypeScript recently added union types [29], and Flow recently added bounded polymorphism [8].

When retrofitting a type system, there is generally a wealth of existing code to which one would like to introduce types. When considering a type system feature, it would be very helpful to know *how* useful the feature would be for typing an existing body of code. Unfortunately, truly gauging the feature's usefulness would require implementing type checking (and possibly inference) for the feature and adding the necessary annotations to the code base. This involves a significant effort, which ultimately may not have lasting value if an alternate type system proves to be more suitable. Some previous work [21, 23, 12, 3] reduces the first burden by providing a reusable framework for implementing type systems, with some inference. However, each still requires manual annotation, so large-scale evaluation remains a substantial undertaking — in one case an expert required over 17 hours to annotate part of one program with type qualifiers [17]. For this reason, in current practice, decisions are often made based on intuition and anecdotal evidence, without the aid of carefully collected quantitative results on the impact of certain features. Anecdotes and other qualitative criteria may validate inclusion of features that are rarely used but important, but quantitative results on a large corpus can rapidly guide the high level design.

In this work, we propose a novel framework, called *trace typing*, for *automatic* and *quantitative* evaluation of retrofitted type system features for existing code bases. Our system works by collecting detailed, unrolled traces of dynamic executions of the code in question. Evaluating a type system feature then requires implementing type checking and inference for the unrolled trace, and also defining a *merge strategy* (explained shortly) for combining the types of different occurrences of a variable in the trace into the corresponding static program entity. The crucial observation here is that type checking and effective inference for the unrolled trace is *far* simpler than static type checking / inference for the full source language, and requires no type annotations due to the simplicity of the trace format. Hence, our system dramatically reduces the implementation effort required to perform quantitative comparisons of different type system variants, and allows type system designers to *automatically* gather feedback from large code bases by typing the trace.

While type inference and checking for unrolled traces is simpler to implement than a static type checker, the ultimate goal of the designer is a type system for source programs, not traces. Our key insight is that–given types for the unrolled trace–many source type systems can be expressed via a family of *merge* operations, used to merge the types of runtime entities that the source type system cannot distinguish. For example, a merge operator can control whether assigning an integer and an object into the same variable yields a union

type for the variable, or a type error. (We give examples of merging throughout the paper.) The type errors found by using merged types give an indicative lower bound on the set of program locations that would be ill-typed under a static type checker, as long as the merge operator produces useful types. A designer can, therefore, evaluate the usefulness of a proposed static type system via suitable merge operators in the trace typing framework, and can also compare the relative power of different static type systems.

We have implemented our techniques in a system for evaluating type system variants for JavaScript. We evaluated our system on a diverse suite of programs, and evaluated several different variations on a core object-oriented type system. In particular, our experiments yielded quantitative answers to the following questions for our benchmarks:

- How pervasive is the need for supporting union types on top of a type system that supports subtyping?
- If union types are supported, what kind of "tag tests" usually occur to discriminate between branches of a union?
- How pervasive is the need for supporting parametric polymorphism?
- How pervasive is the need for intersection types for functions?

The results are given in Section 6. We believe trace typing gives a quick and easy way to obtain answers to these (and similar) questions on a given code corpus.

We have also used trace typing to guide the design of a type system to support ahead-of-time compilation of JavaScript extending previous work [9]. Trace typing allowed us to validate some of our design choices *without implementing a type inference engine* that works on real code. Moreover, it helped point to where we need to extend the type system. Section 6.4 details this case study.

Trace typing is designed to present a rapid, aggregate, quantitative picture to help a type system designer decide whether to support certain features and prioritize the development of a type checker. When features are not supported, a programmer needing said features must either ignore the warnings from the type system, or refactor her code to fit the type system restrictions. For example, if trace typing shows that parametric polymorphism is rarely needed in the code of interest, then the designer can defer supporting polymorphism; in the interim, developers must work around the limitation, e.g., by code cloning. Of course, a type system designer will need to consider other factors besides the output of trace typing when prioritizing, such as idioms used by critical modules, or the difficulty of converting code to fit within certain restrictions.

Contributions. This paper makes the following contributions:

- We propose *trace typing* as an approach for easily evaluating the usefulness of type system features, allowing large bodies of existing code to inform type system design. Trace typing requires far less effort than implementing various static type checkers, and potentially adding type annotations to large bodies of code.
- Using the trace typing approach, we systematically compare the precision of a number of object type systems for JavaScript on some of the most popular packages for `node.js`, totaling over 50,000 lines of code. We include union types as well as several variants of polymorphism in our study.
- We describe our experience with trace typing to guide and check choices in the design of a type system for use by an optimizing JavaScript compiler.
- We use trace typing to automatically identify tag tests in our large corpus, and analyze the relative frequency of different tests and their structure.

```
1  function f(a) { a.p = 7; }
2  function g(b) { return b; }
3  var x = { p: 3 };
4  var y = { p: 4, q: "hi" };
5  var z = { q: "bye", r: false };
6  f(x);  // f.6: {p: Number} -> void
7  f(y);  // f.7: {p: Number, q: String} -> void
8  g(y);  // g.8: {p: Number, q: String} -> {p: Number, q: String}
9  var w = g(z);  // g.9: {q: String, r: Boolean} -> {q: String, r: Boolean}
10 w.r = true;
```

Figure 1 A JavaScript program to illustrate polymorphism.

2 Trace Typing by Examples

In this section, we show informally how trace typing can be used to carry out different kinds of quantitative experiments relevant when designing a type system. Section 3 gives a more formal description of the framework.

2.1 Polymorphism

Figure 1 gives a small JavaScript example. The program allocates new objects at lines 3–5, and then calls functions `f` and `g`. The `f` function accesses the `p` field, while `g` just returns its parameters. All field accesses are well-behaved: they access pre-existing fields, and fields are only updated with a value of a compatible type. Here, we show how trace typing can can *quantify* the relative effectiveness of two type systems with different levels of polymorphism for handling this code.

We focus on the types observed in the dynamic trace for functions `f` and `g`. For our purpose, these types correspond to the parameter and return types observed per function call site. So, for function `f` we observe the type `{p: Number} -> void` for the call at line 6, and `{p: Number, q: String} -> void` for line 7. Similarly, for function `g` we observe type `{p: Number, q: String} -> {p: Number, q: String}` for the line 8 call, and `{q: String, r: Boolean} -> {q: String, r: Boolean}` for line 9. Note that the types above are not based on inference on the body of functions `f` and `g`.

Type system variants can be distinguished by how their merge strategies combine these types into a single function type each for `f` and for `g`. We consider the following merge strategies. $\mathbb{T}_{sub}$ merges, separately the argument and return types, in each case taking the least upper bound in a suitable subtyping lattice. For our example, this strategy yields the type `{p: Number} -> void` for `f` and `{q: String} -> {q: String}` for `g`.[1] $\mathbb{T}_{poly}$ merges the function types by introducing type variables, arriving at a signature with parametric polymorphism (details in Section 4.3). The type we obtain for `f` is the same as in the case of $\mathbb{T}_{sub}$, but for `g`, it is `(E) -> E`; `E` is a type parameter.

Next, the trace typing system performs type checking using the merged types, and reports the number of type errors to the user. This type error count identifies program locations that a static checker for the type system would be unable to validate, a valuable statistic for determining the usefulness of the variant in practice (fewer errors[2] means that the variant can validate more code patterns).

[1] Note that we did not compute a type for `g` via least-upper-bound with respect to *function* subtyping (with contravariance in argument position); see Section 3.1 for further details.

[2] Throughout this paper, we use *error* as a synonym for *static type error*, not to refer to actual developer mistakes in a program.

```
1 function f(x){
2   ... x ... // x.2: A | int
3   if(x instanceof A){ // x.3: A | int
4     ... x ... // x.4: A
5   }
6   ... x ... // x.6: A | int
7 }
8 f(new A());
9 f(3);
```

Figure 2 Tag test example.

With $\mathbb{T}_{sub}$, the types are sufficient for type-checking the bodies of the functions, but g's type is still insufficient to typecheck the dereference at line 10, leaving one error remaining. With $\mathbb{T}_{poly}$, the types are sufficient to ensure there is no type error in the program. The type checking phase also checks conformance of the actual parameters being passed to function calls at lines 6, 7, 8 and 9; these checks pass for both $\mathbb{T}_{sub}$ and $\mathbb{T}_{poly}$.

$\mathbb{T}_{sub}$ produces one type error on this trace, while $\mathbb{T}_{poly}$ produces 0. These counts are underestimates with respect to a (hypothetical) static type checker, both due to our reliance on dynamic information, and due to the over-approximations inherent in any static type checking. Nevertheless, across a large code base, such data could be invaluable in finding the extent to which a particular type system feature, e.g. parametric polymorphism, is useful. Section 6.2 presents results from an experiment that evaluated the usefulness of parametric polymorphism and a synthetic type system with "unbounded" polymorphism on a suite of benchmarks, via comparison of error counts.

2.2 Discriminating Unions

Suppose we wish to design a type system for JavaScript that is equipped to reason about tag checks at the points where the program refines a union type. Such a construct is shown in the example of Figure 2. In this example, a union type with more than one case is narrowed to a specific case guarded by a condition. In general, the guard can be an arbitrary expression, but that makes type checking difficult — it would need to track an arbitrary number of predicates in carrying out an analysis. For example, the following "non-local" variation of the type guard in the example in Figure 2 would be more complex to handle:

```
1   var y = x instanceof A
2   ...
3   if (y) { ...
```

It is simpler to support a limited number of syntactic forms. Can a type system designer obtain *quantitative* feedback on what kind of conditional constructs occur in the code corpora of interest? Are non-local type guards common? What about conjunctions and disjunctions of predicates? Simple syntactic pattern matching is an unreliable way to find this out, because it is fundamentally a flow analysis problem.

Trace typing makes this experiment easy. Trace typing ascribes a type to each occurrence of each variable in the program based on dynamic observation. Given these types, one can find pairs of successive reads of a variable without an intervening write, where the type ascribed to the variable in the second read is different from that of the first. This is a heuristic to identify instances of a variable's type becoming narrower on an unrolled path, likely to be a tag test.

Consider the example in Figure 2 again. We examine some of the type ascriptions carried out by trace typing during an execution of the program. Here, at line 2, `x` refers to an object of type `A` and an `int`, respectively, in the invocations from line 8 and line 9. Let us label the

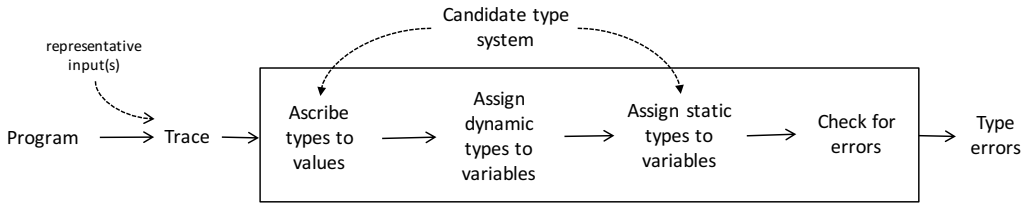


Figure 3 Trace typing framework overview.

two dynamic occurrences of `x` as `x.2.1` and `x.2.2`, where the first index refers to the line on which the variable occurs and the second refers to the invocation sequence number. The same types will be observed for `x.3.1` and `x.3.2`. At line 4, `x.4.1` observes `A`; that is the only dynamic occurrence of `x` on this line. Finally at line 6, `x.6.1` and `x.6.2` observe `A` and `int` respectively.

Crucially, trace typing can be configured to *merge* the types ascribed to variable occurrences on the same line, but across different invocations. Say that our type system includes union types. In that case, `x.2` (the result of merging `x.2.1` and `x.2.2`) gets the type `A | int`, which represents a union type consisting of types `A` and `int`. For `x.3` we get `A | int` as well. However, for `x.4`, we get just `A`. Hence, for successive reads `x.3` and `x.4` (with no intervening write), we observe the situation that the second type is a *refinement* of the first. This is a clue that the conditional at line 3 (that falls between the first and the second reads) is a type guard.

Using this technique on a large code corpus, we found that while non-local type guards are uncommon, boolean combinations of predicates are commonly used in type guards. Section 6.3 details the experiment we carried out to find the nature of tag tests in real code.

3 Trace typing

Figure 3 gives an overview of the trace typing process. The trace typing framework takes two inputs:

- a type system of interest, specified by defining various parameters in a framework (Section 3.1); and
- a program `P` of interest, and inputs for exercising `P`.

Given these inputs, trace typing proceeds in the following four phases:

1. An unrolled *trace* is produced by executing an instrumented version of `P` on the provided inputs (Section 3.2).
2. The types of all values (including objects and functions) and variables in the trace are inferred (Section 3.3).[3]
3. The inferred types are *merged*, coarsening the precise types in the trace to a corresponding source typing (Section 3.4).
4. The resulting types are used for type checking, to estimate the ability of the type system to handle the observed behaviors (Section 3.5).

The number of errors reported in the last step is a useful metric for evaluating type system features (see Section 7 for further discussion). Our framework has proven to be sufficiently expressive to model even modern industrial proposals for retrofitted type systems (Section 4.4). Also, the computed types can be used as part of experiments to test richer type system features, such as tag tests (Section 4.2).

[3] In the remainder of the paper, we use the terms “ascribed” and “inferred” interchangeably.

3.1 Trace Type Systems

Here, we describe the interface by which a type system is specified to the trace typing framework. A type system $\mathbb{T}$ has five components:

$\mathbb{T}.\alpha$**: (Value)** $\rightarrow$ **Type** ascribes a type to a concrete value. For JavaScript, values include both primitives as well as object instances.

$\mathbb{T}.\sqcup$**: (Type, Type)** $\rightarrow$ **Type** computes the least-upper-bound of the input types in the subtyping lattice for the type system. For termination, we require $\mathbb{T}.\sqcup$ to be monotonic.

$\mathbb{T}.\vec{\sqcup}$**: (Function-type...)** $\rightarrow$ **Function-type** takes a list of types for individual invocations of a function `f` and computes a type that can accommodate all invocations. A type for an invocation is computed by applying $\mathbb{T}.\alpha$ to the receiver, argument, and return values, and then combining them into a function type. $\mathbb{T}.\vec{\sqcup}$ often differs from applying $\mathbb{T}.\sqcup$ to the invocation types; further discussion below.

$\mathbb{T}$`.equiv`**: (`FI` | `FS`)** $\times$ **(`CI` | `CS`)** governs how types for different trace occurrences of the same source variable are merged to approximate source-level typing (see Section 3.4). The first component determines whether to maintain a separate type for a local variable at different program points (flow sensitivity), while the second component determines if separate types should be maintained across different function invocations (context sensitivity). Note that use of `CI` vs. `CS` is usually dictated by the choice of $\mathbb{T}.\vec{\sqcup}$ (see Section 6.2). Also, our implementation supports more fine-grained flow- and context-sensitivity settings; here we only present two options for simplicity.

$\mathbb{T}$`.check`**: (TypeEnv, Statement)** $\rightarrow$ **number** is a type checking function that counts the number of type errors a trace statement causes in a given type environment.

Together, $\mathbb{T}.\sqcup$, $\mathbb{T}.\vec{\sqcup}$, and $\mathbb{T}$`.equiv` comprise the type system's *merge strategy*. **Value** represents the values of our trace language (see Section 3.2), while **Type** and **Function-type** are types defined entirely by $\mathbb{T}$. Defining $\mathbb{T}.\alpha$ for JavaScript objects is non-trivial, since the shape and property types of the object may change over time; we detail the issues involved in Section 4.1.

Note that $\mathbb{T}.\vec{\sqcup}$ should *not* use contravariance when handling function parameters, unlike typical function subtyping. $\mathbb{T}.\vec{\sqcup}$ is used to compute a type τ for function `f` that is consistent with all observed invocations of `f`. Hence, each parameter type in τ should accommodate *any* of the corresponding parameter types from the individual calls, a covariant handling. For an example, see the handling of calls to `g` in Figure 1 (Section 2.1).

Example 1: Simple OO. To simulate a basic structural OO type system with width subtyping:

- $\mathbb{T}.\alpha$ gives a type to an object by recursively characterizing the types of its properties (see Section 4.1 for details).
- $\mathbb{T}.\sqcup$ gives the least-upper-bound of two object types (retaining common fields that have identical types) and returns the same type if given two identical types. Otherwise it returns $\top$.
- $\mathbb{T}.\vec{\sqcup}$ applies $\mathbb{T}.\sqcup$ to the individual argument and return types (*not* using standard function subtyping; see above) to compute a generalized function type.

⟨*trace*⟩ ::= ⟨*statement*⟩*
⟨*statement*⟩ ::= ⟨*var*⟩ = ⟨*expression*⟩ | ⟨*var*⟩.⟨*name*⟩ = ⟨*var*⟩ | delete ⟨*var*⟩.⟨*name*⟩ | ⟨*meta*⟩
⟨*expression*⟩ ::= ⟨*var*⟩ | ⟨*var*⟩.⟨*name*⟩ | `allocate` | `null` | `undefined` | ⟨*bool*⟩ | ⟨*number*⟩ | ⟨*string*⟩
⟨*var*⟩ ::= `v1` | `v2` | `v3`, ...
⟨*name*⟩ ::= a | b | ...
⟨*meta*⟩ ::= `begin-call`⟨*var*⟩⟨*var*⟩⟨*var*⟩* | `end-call`⟨*var*⟩ | `end-initialization`⟨*var*⟩

Figure 4 TL grammar.

- $\mathbb{T}$.`equiv` is (`FI`,`CI`).
- $\mathbb{T}$.`check` performs standard assignment compatibility checks for variable and field writes, property type lookup on property reads, and standard checks at function invocations.

Example 2: Union Types. To extend Example 1 with union types, we modify $\mathbb{T}.\sqcup$ to introduce unions when merging different sorts of types (e.g., a number and object type). For the code '`var x = 3; x = { f : 4 };`', `x` would be given the union type `number` | {f : `number`}: Section 4.1 discusses how we perform type checking in the presence of such union types.

Example 3: $\mathbb{T}_{sub}$ and $\mathbb{T}_{poly}$ revisited. We referred to $\mathbb{T}_{sub}$ and $\mathbb{T}_{poly}$ in Section 2.1. $\mathbb{T}_{sub}$ is precisely the Simple OO of Example 1. $\mathbb{T}_{poly}$ replaces $\mathbb{T}.\vec{\sqcup}$ of $\mathbb{T}_{sub}$ to introduce parametric polymorphism (described further in Section 4.3).

3.2 Traces

The execution of a source code program can be recorded as a finite sequence of primitive instructions that reproduces the dataflow of the execution. We call such a sequence a trace. A trace is a simplified version of the original source program:

- Every dynamic read or write of a variable or subexpression is given a unique identifier.
- Control flow structures are erased: conditionals are reduced to the instructions of the conditional expression and chosen branch, loops are unrolled, and calls are inlined.
- Native operations are recorded as their results; e.g. a call to the runtime's `cosine` function is recorded as the computed value.

We express our JavaScript traces as programs in a simple language TL. TL is an imperative, object-oriented language without control flow structures or nested expressions. TL has reads and writes for variables and fields, deletion for fields, and values (objects and primitive values). The syntax of TL can be seen in Figure 4.

Figure 5 shows excerpts from the trace for executing the code in Figure 1. Many JavaScript details are omitted here for clearer exposition but handled by our implementation. The variable names in the trace are generated from source variable names where present, and otherwise fresh temporaries are generated. (Note that fresh temporaries are used within each call to `f`.) A source mapping for each trace statement is maintained separately.

The trace contains some meta-statements to aid in later analyses. Each call is demarcated by `begin-call` and `end-call` statements, to aid in recovering the relevant types for the invocation. Data flow from parameter passing and return values is represented explicitly. The `end-initialization` statement marks the end of a constructor or initialization of an object literal.

```
1  <tmp0@G> = allocate // ... allocation and assignment at line 6
2  <tmp1@G> = 3
3  <tmp0@G>.p = <tmp1@G>
4  end-initialization <tmp0@1>
5  <x@G> = <tmp0@G>
6  begin-call <f@G> <x@G> // ... first call to f, at line 9
7  <a@f_1> = <x@G>
8  <tmp0@f_1> = 7
9  <a@f_1>.p = <tmp0@f_1>
10 end-call
11 begin-call <f@G> <y@G> // ... second call to f, at line 10
12 <a@f_2> = <y@G>
13 <tmp0@f_2> = 7
14 <a@f_2>.p = <tmp0@f_2>
15 end-call
```

Figure 5 Excerpts from the trace for the program in Figure 1. The name of variables include their scope, e.g. `<x@G>` is the `x` variable in the global scope, and `<a@f_2>` is the `a` variable during the second call to `f`.

3.3 Initial Type Ascription

Given a trace and a type system as specified in Section 3.1, trace typing first ascribes precise types to variables and values in the trace, without consideration for mapping types back to the source code. First, types are ascribed to all values. Primitive values and objects are handled directly using $\mathbb{T}.\alpha$, while functions are handled by combining the type for each invocation with $\mathbb{T}.\vec{\sqcup}$, as described in Section 3.1.

Once types for values have been ascribed, each trace variable is given the type of the value it holds (recall that each trace variable is only written once). For a given trace T, we define the initial type environment Γ_0 as: $\forall v \in \mathsf{Variables}(T).\, \Gamma_0(v) = \mathbb{T}.\alpha(\mathsf{ValueOf}(T, v))$. This typing for trace variables is very precise—every variable access is given its own type, based solely on the value it holds at runtime. The next phase (Section 3.4) generalizes these initial types to more closely mimic the desired source-level treatment of variables.

Example. Consider the variable `<tmp1@G>` at line 2 in Figure 5; it gets the value `3`, for which $\mathbb{T}_{sub}.\alpha$ yields the type `Number`. Similarly the type of `<x@G>` at line 5 is `{p: Number}` because that is the type ascribed to the value in `<tmp0@G>` at line 5.

3.4 Type Merging and Propagation

To model realistic type systems, we construct a less precise type environment $\hat{\Gamma}$ by merging the types of "equivalent" trace variables in Γ_0. Variable equivalence classes are determined by the $\mathbb{T}$.`equiv` operator provided by the type system (see Section 3.1). If the first component of $\mathbb{T}$.`equiv` is `FI`, we employ a flow-insensitive treatment of variables: all variable occurrences `v_i` within a single dynamic call corresponding to the same source variable `v` are made equivalent. No such equivalences are introduced if the first $\mathbb{T}$.`equiv` component is `FS`. If the second component of $\mathbb{T}$.`equiv` is `CI`, we impose a context-insensitive treatment of variables: each matching `v_i` *across* function invocations (i.e., the same read or write of source variable `v`) is placed in the same equivalence class.

Given variable equivalence classes and Γ_0, Figure 6 defines $\hat{\Gamma}$ in terms of two components: $\hat{\Gamma}_m$ for *merging* equivalent variables, and $\hat{\Gamma}_p$ for *propagating* across assignments. (As $\hat{\Gamma}_m$ and $\hat{\Gamma}_p$ are themselves defined in terms of $\hat{\Gamma}$, the equations must be solved by computing a fixed point.) Given trace variable x, $\hat{\Gamma}_m(x)$ computes the least-upper bound ($\mathbb{T}.\sqcup$) of all variables in its equivalence class $\hat{x}$. However, this merging alone is insufficient for mimicking

$$\hat{\Gamma}(x) = \hat{\Gamma}_m(x)\ \mathbb{T}.\sqcup\ \hat{\Gamma}_p(x) \qquad\qquad \hat{\Gamma}_m(x) = \mathbb{T}.\bigsqcup_{x_i \in \hat{x}} \hat{\Gamma}(x_i)$$

$$\hat{\Gamma}_p(x) = \mathbb{T}.\bigsqcup_{'x=rhs' \in stmts} \begin{cases} \Gamma_0(x), & rhs = \text{allocate} \vee rhs \in \textit{primitives} \\ \hat{\Gamma}(y), & rhs = y \\ \hat{\Gamma}(b).p, & rhs = b.p \wedge \hat{\Gamma}(b) \text{ is object with property } p \\ \Gamma_0(b).p, & rhs = b.p \wedge (\hat{\Gamma}(b) \text{ not an object} \vee \hat{\Gamma}(b).p \text{ not present}) \end{cases}$$

Figure 6 Equations for type merging and propagation.

```
 1: f :: <({p: Number}) -> Undefined>
 2: g :: <({q: String}) -> {q: String})>
 6: x :: {p: Number}
 7: y :: {p: Number, q: String}
 8: z :: {q: String, r: Boolean}
12: w :: {q: String}
```

Figure 7 Types ascribed for variable writes in Figure 1 with $\mathbb{T}_{sub}$.

source typing, as it does not consider relationships between variables: if the program contains statement `x = y`, then the type of `y` influences the type of `x` in a source level type system.

$\hat{\Gamma}_p$ in Figure 6 handles type propagation across assignments. The first case handles assignments of values, using the baseline environment Γ_0. For the second case, $x = y$, we generalize $\hat{\Gamma}(x)$ to include $\hat{\Gamma}(y)$. This step is important to mimic source-level handling of the assignment. The final two cases handle object field reads $b.p$. If $\hat{\Gamma}(b)$ is an object type with a property p, the handling is straightforward. However, this may not hold, due to other approximations introduced in computing $\hat{\Gamma}(b)$ (e.g., if p were dropped when merging object types due to width subtyping). If $\hat{\Gamma}(b).p$ does not exist, we fall back to $\Gamma_0(b).p$, i.e., we use the *precise* type of b. Without this treatment, type errors could propagate throughout the remainder of the trace, misleadingly inflating the type error count. Our fallback to the precise type of b helps to localize type errors, thereby allowing for gathering more useful information from the rest of the trace than the alternative of producing $\top$. For example, if `x` holds a number and later an object with property `foo`, `FI` merging will give it type $\top$. A read of `x.foo` would produce a type error. Rather than tainting all subsequent uses of the read result, we ascribe the type of the value observed as the result of that read.

Example. Consider type propagation for the `a` variable in function `f` of Figure 1, using type system $\mathbb{T}_{sub}$ from Section 2. The corresponding variables in Figure 5 are `<a@f_1>` and `<a@f_2>`, whose respective types are `{p: Number}` and `{p: Number, q: String}` in Γ_0 (`<y@G>`'s initialization (line 4 in Figure 1) is elided in Figure 5). Hence, with the width-subtyping-based merge mechanism $\mathbb{T}_{sub}.\sqcup$, both `<a@f_1>` and `<a@f_2>` are assigned type `{p: Number}` in $\hat{\Gamma}$. The types in $\hat{\Gamma}$ for all variables in Figure 1 (using $\mathbb{T}_{sub}$) appear in Figure 7.

3.5 Type Checking

Once type propagation is done, type checking can be performed applying $\mathbb{T}$.`check` to every statement in the trace, using $\hat{\Gamma}$ as the context. $\mathbb{T}$.`check` should return, for each statement, the number of falsified antecedents for the corresponding type rule (returning 0 implies a well-typed statement).

$$
\begin{array}{rcl}
\tau & ::= & \top \mid \bot \mid \mathsf{number} \mid \mathsf{boolean} \mid \mathsf{string} \mid \mathbb{O} \mid \mathbb{F} \mid \bigcup \overline{\tau} \mid \mu X.\tau \\
\mathbb{O} & ::= & \mathit{PropName} \to \tau \\
\mathbb{F} & ::= & [\tau](\overline{\tau}) \to \tau \mid \bigwedge \overline{\mathbb{F}}
\end{array}
$$

Figure 8 Types used in the type systems we model.

Note that our type propagation biases the location of type errors to occur more often at reads instead of writes. Type propagation uses the $\mathbb{T}.\sqcup$ operator to find a variable type that handles all writes to that variable. This process may produce a type higher in the subtyping lattice than what the programmer may have intended, causing type errors to be pushed to uses of that variable.

Consider the example: '`x = {name: "Bob"}; x = {nam: "Bill"}; print(x.name);`' Most likely, the programmer intended the type of `x` to include a `name` field. If this type were declared, the type checker would produce an error at statement two, due to the misspelled `nam` field. With trace typing under the $\mathbb{T}_{sub}$ type system, we will instead compute a type for `x` that has *no* fields by merging the types of the two objects — essentially over-generalizing due to the presence of what is well described as a static type error. Hence, the type error is pushed to the access of `name` at statement three, which is no longer in the type of `x`.

Intuitively, we want that when mimicking two comparable source-level type systems, trace typing ends up reporting fewer type errors for the more powerful one. Indeed, our results show that this is generally the case. However, this is not *guaranteed* to be the case because our ascription operation ($\mathbb{T}.\alpha$) can sometimes over-generalize, leading to unpredictable error counts. See Section 7 for further discussion of this matter.

4 Instantiations

In this section, we detail several instantiations of the trace typing framework of Section 3, showing its flexibility. We first describe a core type system that allows many of JavaScript's dynamic object behaviors. We then show how this system can be used to detect possible tag tests in programs. Then, we present an extension of this system to test the usefulness of parametric polymorphism. Finally, we show an instantiation with a different, stricter type system for objects based on recent work [9], and how to evaluate various features of that system using trace typing.

4.1 Core Type System

The types we use for most of our experiments appear in Figure 8. The top ($\top$), bottom ($\bot$), primitive, union ($\bigcup$), and recursive types (μ) are standard. An object type ($\mathbb{O}$) is a map from property names to types. Function types ($\mathbb{F}$) are either standard (with receiver, argument, and return types), or an intersection type over function types. We use intersection types as a very precise model of call-return behavior, for a limit study of the usefulness of polymorphism (Section 6.2).

In our implementation, our object types are significantly more complex, mirroring the many roles objects play in JavaScript. In JavaScript, functions are themselves objects, and hence can have properties in addition to being invokable. Arrays are also objects with numeric properties and possibly non-numeric as well. Our implemented object types model these semantics, but we elide them here for simplicity. JavaScript also has prototype inheritance, where a read of property p from object o is delegated to o's prototype if p is not present

$$\begin{aligned} \mathit{shapeMap}(o) &\equiv \{(p_i, v_i) \mid o.p_i = v_i \text{ at some trace point}\} \\ \mathbb{T}.\alpha(o) &\equiv \{(p_i, \tau_i) \mid M = \mathit{shapeMap}(o) \wedge p_i \in \mathit{dom}(M) \wedge \ \tau_i = \bigsqcup_{v \in M[p_i]} \mathbb{T}.\alpha(v)\} \end{aligned}$$

Figure 9 Type ascription for objects.

on o. For the core type system, object types collapse this prototype chain, so unshadowed properties from the prototype parent are collapsed into the child object type (more discussion shortly). The type system of Section 4.4 deals with prototype inheritance more precisely.

Ascribing Object Types. As noted in Section 3.1, defining the $\mathbb{T}.\alpha$ operation for object values is non-trivial. Since object properties may point to other objects, types for object values are inter-dependent. Further, JavaScript objects may be mutated in many ways: beyond changing property values, properties can also be added and deleted. Even the prototype of an object may be mutated.

Trace typing is currently limited to object types that do not change over time, i.e., only one type can be associated with each object instance during the entire execution. So, for an object o, $\mathbb{T}.\alpha(o)$ must merge together the different observed properties and property types for o into a single object type. Nearly all practical type systems use object types in this manner, as state-dependent object types require complex reasoning and restrictions around pointer aliasing.

Figure 9 gives our definition of $\mathbb{T}.\alpha$ for objects. We first define $\mathit{shapeMap}(o)$, a map from property names to sets of values (shown as a set of pairs). $\mathit{shapeMap}(o)$ captures all the property values that could be read from o at any point in the trace. (Note that we do not require the property read to actually exist in the trace.) Since JavaScript property reads may be delegated to the prototype object, $\mathit{shapeMap}(o)$ gives a "flattened" view of o that includes inherited properties. Consider the following example:

```
1 var y = {};
2 var x = { a: 4 } proto y;  // prototype inheritance shorthand from [9]
3 x.b = { c: false };
4 y.d = "hello";
5 x.a = 10;
```

For this example, the shape map for the object allocated on line 2 is: $[\texttt{a} \mapsto \{4, 10\}, \texttt{b} \mapsto \{\{\texttt{c} : \texttt{false}\}\}, \texttt{d} \mapsto \{\text{"}\texttt{hello}\text{"}\}]$.

The object type $\mathbb{T}.\alpha(o)$ is written as a set of pairs in Figure 9. The properties in $\mathbb{T}.\alpha(o)$ are exactly those in the domain of $\mathit{shapeMap}(o)$. The type for property p_i is obtained by applying $\mathbb{T}.\alpha$ to each value of p_i in $\mathit{shapeMap}(o)$, and then combining the resulting types using $\mathbb{T}.\sqcup$. When computing $\mathbb{T}.\alpha$, cyclic data structures must be detected and handled by introducing a recursive type.

Type Checking. The type rules we checked in our implemented type systems are standard. Property accesses can be checked normally, since the accessed property name is always evident in the trace. Function invocations are checked normally with one caveat: passing too few arguments is permitted, as all JavaScript functions are implicitly variadic.

For type systems with union types, we optimistically allow most operations that work for one case of the union (e.g., property access on a union typed expression) without validating that the code has checked for the appropriate case, but treat assignment of union-typed expressions to storage locations soundly (TypeScript and Flow have very similar rules for

unions). Our trace language does not currently include conditionals, making it non-trivial to use tag tests to eliminate union types [20, 28]. However, our types do allow for discovering likely tag tests in the program, as we show below.

In the case when $\mathbb{T}.\sqcup$ causes a type to be $\top$, we count every use of and assignment to a variable of that type to be an error.

4.2 Detecting Tag Tests

To soundly eliminate union types, a type system must support narrowing the union under a conditional that checks some tag information of a value. While the theory for such narrowing is rich and well-developed [20, 28], industrial type systems for JavaScript (Flow and TypeScript) treat union elimination unsoundly in the same "optimistic" manner we do (Section 4.1). A key issue is that tag tests can vary in complexity, from simple type checks (e.g., JavaScript's `typeof` operator) to complex combinations of user-defined predicates, and it is unclear what level of complexity must be handled to provide adequate support for tag tests.

Using trace typing, we designed a technique to observe which tag test constructs are being used most frequently. As described previously in Section 2.2, the technique works by observing when there are two consecutive reads of a variable `v` (with no intervening write), and `v`'s type is narrower at the second read than the first. This technique requires using the `FS` setting in $\mathbb{T}$.`equiv`, to keep separate types for each variable occurrence.[4] For function types, we used $\mathbb{T}.\overrightarrow{\sqcup}$ from $\mathbb{T}_{sub}$, thereby generalizing using only subtype polymorphism. In Section 6.3, we discuss the kinds of tag tests discovered in our benchmarks with this technique and their prevalence.

4.3 Parametric Polymorphism

Here, we define $\mathbb{T}.\overrightarrow{\sqcup}_{poly}$, an instantiation of the $\mathbb{T}.\overrightarrow{\sqcup}$ operator that discovers types with parametric polymorphism. With this approach, one can evaluate the usefulness of adding parametric polymorphism to a retrofitted type system. $\mathbb{T}.\overrightarrow{\sqcup}_{poly}$ works by enumerating all possible type parameter replacements in each invocation type for a function, and finally choosing the most general parameterized type matching all invocations. The enumeration proceeds by replacing all subsets of occurrences of a concrete type by a type variable. Consider the identity function, `function id(x){ return x; }`, and two invocations, `id(3)` and `id(true)`. The observed invocation types for `id` are `Number -> Number` and `Boolean -> Boolean`. (Applying $\mathbb{T}.\overrightarrow{\sqcup}$ from $\mathbb{T}_{sub}$ to these types would yield the unhelpful type $(\top) \to \top$ for `id`.) For the first type, the enumeration would yield types `X -> Number`, `Number -> X`, and `X -> X` (along with the original type). After proceeding similarly for the second type, $\mathbb{T}.\overrightarrow{\sqcup}_{poly}$ returns the type `X -> X` for `id`, as it is the most general type matching both invocations.

In general, $\mathbb{T}.\overrightarrow{\sqcup}_{poly}$ also generates signatures with multiple type parameters. It also attempts to replace types nested one level into each argument or return type, to discover types operating over generic data structures. While this enumeration is exponential in the arity of the function, we did not observe an appreciable slowdown in practice.

After discovering polymorphic types, we must check the invocations against those types in $\mathbb{T}$.`check`. The checking can be done by treating each type variable as a direct subtype

[4] Our implementation also disabled the assignment merging shown in Figure 6, to discover more tag tests.

of $\top$, distinct from other type variables. Our implementation currently uses an alternate strategy of modifying the context-sensitivity policy in $\mathbb{T}$.`equiv`, so that variables in function invocations with distinct instantiations of the type parameters are not merged. This strategy corresponds to checking a more powerful form of bounded polymorphism, which may yield a reduced error count compared to directly checking unbounded type parameters.

4.4 Fixed Object Layout

We now show how to enhance the object types presented in Section 4.1 to support *fixed object layout*, as described in recent work by Choi et al. [9]. In general, JavaScript objects is used as dictionaries, with arbitrary addition and deletion of properties (keys). However, implementing all objects in this manner would lead to poor performance. Real-world programs *often* do not use this dictionary-like behavior. Objects often behave like records, in which the set of properties is stable: after initialization, new properties are not added or removed. Modern just-in-time compilers detect the record usage pattern at run time, and use it to optimize the object representation where possible.

Choi et al. [9] show how to enforce the record usage pattern in a type system, enabling efficient static compilation of a JavaScript subset. Prototype inheritance complicates the design of such a type system, since a write of an inherited property creates a shadowing property in the child object. Consider the following example:

```
1 var o1 = {a:3, m:function(x) {this.a = x;}}
2 var o2 = {b:5} proto o1 // prototype inheritance shorthand from [9]
3 o2.a = o2.b; // adds field a to o2
```

The write at line 3 adds a local `a` property to `o2`, rather than updating the inherited property `o1.a`. The type system for fixed layout handles this case by distinguishing between *read-write* properties local to an object and *read-only* properties inherited from the prototype chain. With this distinction, the type system would reject the statement on line 3 above, as it is a write to a read-only property.[5]

To track read-only and read-write properties in object types, we extend $\mathbb{T}.\alpha$ from Section 4.1 as follows. Instead of tracking a single shape map for each object, we keep a read-write shape map for locally-declared properties, and a read-only shape map for inherited properties. In Section 4.1, the domain of $shapeMap(o)$ includes any p_i present on o at any point in the execution. Here, to enforce the fixed layout property, we restrict the domain of each shape map to properties present before the end of o's initialization. (Recall from Section 3.2 that our traces include `end-initialization` statements to mark these points.) Given these two shape maps, computation of read-only and read-write properties in each object type proceeds as in Figure 9. Type checking is enhanced to ensure that property writes are only performed on read-write properties.

We also check for two additional properties from the Choi et al. type system [9]. The type system requires any object used as a prototype parent to have a *precise* type, i.e., no properties can have been erased from the type using width subtyping. (This is required to handle subtle cases with inheritance [9].) In trace typing, we check this property by only flagging types of object literals as precise, so if width subtyping is ever applied, the resulting type is not flagged. Finally, the type system requires that if a property from a prototype

[5] The type system of Choi et al. also tracks which properties are written on method receivers, to locally check validity of inheritance [9]. With trace typing, this information need not be tracked explicitly, since receiver types are propagated into method calls.

is shadowed, the parent and child properties have the same type. This condition is easily checked in $\mathbb{T}.\alpha$ once read-only and read-write properties are computed.

5 Implementation

We implemented the trace typing process described in Section 3 as a toolfor typing JavaScript programs. This section reports on some of the more noteworthy aspects of the implementation.

Tool Architecture. Our tool consists of two components: Trace Collector and Trace Typer. Trace Collector is a Jalangi [26] analysis that obtains a trace of a program of interest by monitoring the execution using source-level instrumentation; it consists of about 2500 lines of JavaScript code. Trace Typer implements the core trace typing framework. The framework and the type systems described in Section 4 are implemented in about 4000 lines of TypeScript, of which about 1000 lines are the type system implementations.

Modeling JavaScript Semantics. In our implementation, great care is taken to accurately model JavaScript's complex primitive operators. Much of the complexity in operator semantics lies in the implicit conversions performed on operands. For example, the binary `&&` operator first coerces each of its operands to a boolean value; this coercion can be applied to a value of any type. However, the value of the `&&` expression is always one of the *uncoerced* operands. The implementation creates trace statements that models these conversions explicitly, enabling accurate data flow tracking and type ascription.

Modeling the Native Environment. To handle interactions with the native environment (e.g., built-in JavaScript library routines), we require a model of the native behavior. We express native models using trace statements, avoiding the need for a separate modeling language. In many cases, we can actually infer native models from the concrete states of the execution, thereby avoiding the significant work of writing models by hand. Model inference works by associating a new global *escape variable* with each object that can escape to the native environment. Passing an object to the native environment is modeled as a write to the corresponding escape variable, while retrieving the object from the environment is modeled as a read. With this handling, a model can be automatically inferred for much of the native environment, including complex functions such as the overloaded Array constructor. Our technical report [4] provides more details.

6 Experiments

We report on several experiments we conducted using trace typing. First, we studied the trends in error counts for six type system variants, generated by varying handling of subtyping and function types (Section 6.2), yielding useful insights on the relative importance of these variants. Second, we used trace typing to discover tag tests in our benchmark and characterize their relative frequency and complexity (Section 6.3). Finally, we studied several questions around the restrictiveness in practice of a type system for fixed object layout (Section 6.4). Together, these experiments show the usefulness and versatility of the trace typing framework. We first describe our benchmarks, and then present the experiments.

Table 1 Benchmarks sizes and error rates for the fixed-object-layout type system. 'LOC' is the lines of code in the program. 'LOC exec' is the lines covered in our execution of the program. 'Length' is the total number of trace statements. 'ro/rw', 'prototypal' and 'inheritance' are the error-rates for different each kind of type check.

	Benchmark sizes			Fixed-object-layout error rates		
Benchmark	**LOC**	**LOC exec**	**Length**	**ro/rw**	**prototypal**	**inheritance**
`escodegen`	2132	723	375325	0/53	0/0	0/217
`esprima`	4610	1052	23986	0/44	0/2	0/298
`lazy.js`	2557	1016	25439	114/387	0/17	2/470
`minimist`	186	151	140812	0/14	0/0	0/81
`optparse`	222	141	15246	0/15	0/1	0/51
`qs`	726	256	102637	0/41	0/0	0/62
`typescript`	35456	8139	167730	1/2085	2/3	0/3286
`underscore`	1098	727	120081	2/175	0/0	1/294
`xml2js`	840	275	73672	0/42	1/2	0/91
Total	52193	14541	1243857	117/2856	3/25	3/4850

6.1 Benchmarks

We use nine popular packages from `npm`, the node.js package manager, as benchmark programs. Table 1 measures the static source code size of each program,[6] as well as the number of statements in the traces we generate and the number of static source lines covered by our executions. `underscore` and `lazy.js` are utility libraries. `esprima` and `escodegen` are a JavaScript parser and code generator, respectively. `typescript` is the compiler for the TypeScript language.[7] `minimist`, `optparse` and `qs` are parsers for command line options and query strings. `xml2js` is an XML serializer/deserializer.

These benchmarks capture a range of interesting JavaScript styles. The parsers, code generators, and compilers manipulate OO-style AST data structures In contrast, the utility libraries and option parsers are highly reflective (e.g., constructing or extending objects using dynamically-computed property names), but are otherwise written in a largely functional style. Hence, the suite as a whole exercises a variety of features of trace typing.

To exercise the programs for trace generation, we wrote a separate driver for each benchmark. For the option parsers and `escodegen`, this driver comprised extracted tool invocations from their test suites. We exercised the utility libraries with code snippets from their official tutorials.

6.2 Comparing Type System Variants

For our first experiment, we compared six type systems, generated by varying the $\mathbb{T}.\sqcup$ and $\mathbb{T}.\vec{\sqcup}$ operators over the core types of Section 4.1. For $\mathbb{T}.\sqcup$, we used two options:

- **subtyping**: This uses standard structural (width) subtyping for objects, as in $\mathbb{T}_{sub}$ from Section 2.1 (see also Example 1 in Section 3.1).
- **unions**: $\mathbb{T}.\sqcup$ treats two object types as in **Subtyping**, but also introduces a union type when applied to an object type and a primitive (see Example 2 in Section 3.1).

[6] Non-comment, non-blank lines as counted by `cloc`, excluding build scripts, tests and test harness code.
[7] Table 1 gives lines of JavaScript source (the compiler is bootstrapped).

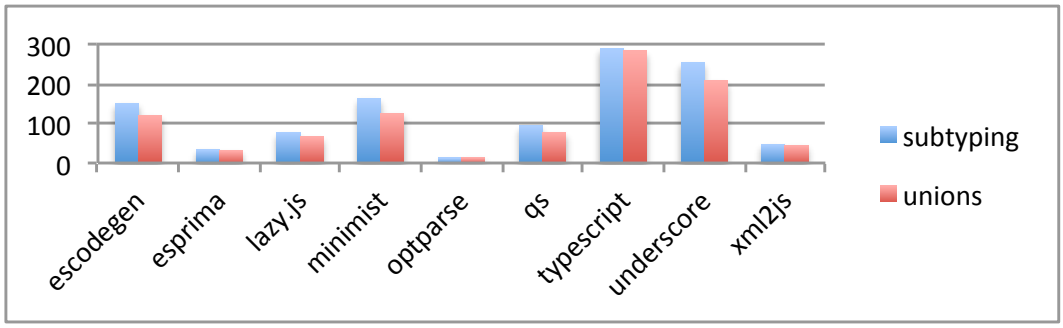


Figure 10 Error counts for **subtyping** and **unions**, selecting **base** for $\mathbb{T}.\vec{\sqcup}$.

For $\mathbb{T}.\vec{\sqcup}$, we use three options:

- **base**: $\mathbb{T}.\sqcup$ is applied individually to argument and return types, as discussed in Section 3.1.
- **poly**: This uses $\mathbb{T}.\vec{\sqcup}_{poly}$ to discover parametric polymorphism, as discussed in Section 4.3.
- **intersect**: $\mathbb{T}.\sqcup$ simply combines two function types into an intersection of the two types. This strategy yields more precise types than any practical type system, and hence is only used as a limit study of the usefulness of polymorphic function types.

$\mathbb{T}$.`equiv` uses `FI` for flow-insensitive variable types in all configurations. Context sensitivity is varied in accordance with $\mathbb{T}.\vec{\sqcup}$: for **base**, we use `CI`, while for **intersect** we use `CS` (since each invocation may have its own case in the intersection type). Context sensitivity for **poly** was discussed in Section 4.3.

After collecting traces, we ran our trace typer with each of our type system variants, counting the number of static (source location) type errors that arose. We include only type errors that occur in the subject program, not in the libraries they depend on.

Experiments were run on a quad-core Intel Core i7-3520M@2.90GHz with 16GB RAM, running Ubuntu 15.10 and node.js 5.0. Collecting traces and running all six configurations required less than 15 minutes.

Our full data set is available [4]. We present two select slices of data here. Figure 10 presents the number of syntactic locations with errors for **subtyping** and **unions**, where $\mathbb{T}.\vec{\sqcup}$ is **base** in either case. Figure 11 shows error counts for the three $\mathbb{T}.\vec{\sqcup}$ settings, where $\mathbb{T}.\sqcup$ is **subtyping** in all three cases. The trends shown here are similar across other configurations.

Overall we see trends we would expect, with more powerful type systems yielding fewer errors. There are two small inversions in the results, where a more precise configuration (**poly**) produces more errors than a less precise configuration (**base**) discussed shortly.

This experiment let us find *quantitative* answers to the kinds of questions we raised in Section 1 and Section 2. We remind the reader that the objective of these experiments is not to pass verdict on the usefulness of certain type system features in an absolute sense. Rather the objective is to help designers of retrofitted type systems prioritize features based on empirical evidence rather than by intuition alone. If a type system feature ends up unsupported, then users will need to work around that limitation in their code (or live with type errors).

How prevalent is the need for union types? While Figure 10 shows that union types do reduce error counts over just using subtyping (as expected, since unions are more powerful), the degree of reduction is relatively small. While TypeScript and Flow now both include unions, TypeScript did not add them until two years after its initial release. The small error

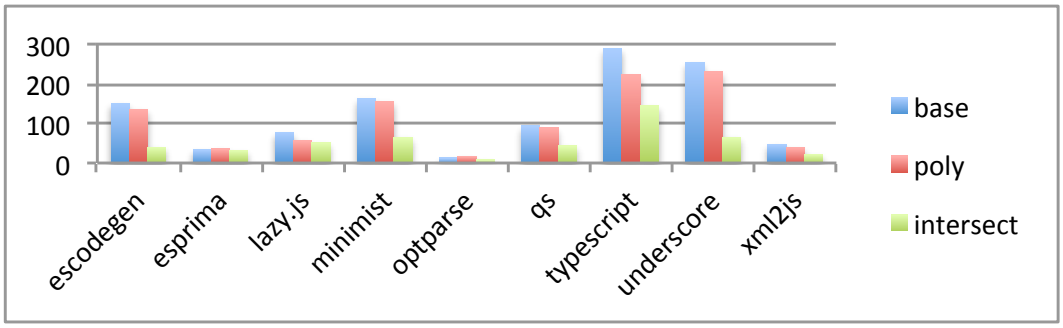


Figure 11 Error counts for different $\mathbb{T}.\vec{\sqcup}$ settings, selecting **subtyping** for $\mathbb{T}.\sqcup$.

count reduction due to union types in our experiments could help to explain why they were not needed in TypeScript from the very beginning.[8]

How prevalent is the need for parametric polymorphism? For intersection types? As shown in Figure 11, the **intersection** configuration always reduced errors compared to **poly**, while **poly** almost always reduces errors compared to **base**[9]. While the error count reductions for **poly** are not dramatic in general (`typescript` is discussed below), parametric polymorphism is crucial for ascribing types to core routines manipulating data structures (e.g., array routines), and hence its inclusion in TypeScript and Flow is unsurprising. The dramatic drop in error count for several programs under the **intersection** configuration gives strong evidence that JavaScript requires types that can express non-uniform polymorphism, such as intersection types. In fact, TypeScript has long supported overloaded functions for expressing intersection types.[10]

The `typescript` benchmark stands out from the others as it was originally implemented in TypeScript, and hence we would expect it to be mostly well typed. Our results confirm this: the error count is only 290 in 8139 (JavaScript) lines of code for the least precise configuration, a much lower error rate than the other programs. Also, the significant reduction in errors going from **base** to **poly** for the benchmark (see Figure 11) corresponds nicely to the use of parametric polymorphism in the TypeScript code base; see Section 7 for an example.

While in the above experiment we used trace typing to retroactively find evidence of the need of type system features, the next two experiments are concerned with more exploratory questions.

6.3 Finding Tag Tests

Here, we provide the first empirical analysis of the use of tag tests in JavaScript which identifies tag tests based on observed narrowing (as described in Sections 2.2 and 4.2), rather than syntactic criteria. We post-process the ascription results to automatically identify occurrences of guard predicates using the techniques of Section 4.2. We then manually analyze the predicates to identify non-local guards (where the boolean result of a check is computed outside a syntactic conditional statement), non-atomic guards (i.e., conjunction and disjunction of guards), and predicate functions (other functions whose result indicates a certain refinement).

[8] TypeScript's issue tracker has further discussion of their motivation for adding unions; see `https://github.com/Microsoft/TypeScript/issues/805`.

[9] The inversions in `esprima` and `optparse` are due mostly to implementation quirks, such as an incomplete merge operator on recursive types that returns $\top$ more than necessary.

[10] See `http://www.typescriptlang.org/Handbook#functions-overloads`.

Table 2 Tag test classification results.

Measure	Count	Example
Type guards	164	-
Non-local	6	`var isObj = typeof x == 'object'; if(isObj)`
Non-atomic	46	`if(typeof x == 'string' || typeof x == 'object')`
typeof	30	`typeof x === 'function'`
instanceof	9	`x instanceof Function`
tag field	16	`x.kind = Kinds.Template`
predicate function	30	`Array.isArray(x)`
property presence	35	`x.hasOwnProperty('prop')`, `x.prop === undefined`
.prototype/.constructor	3	`x.constructor === Function`
.. test on property	37	`typeof x.prop === 'function'`
other	23	`className === toString.call(x)`

Methodology. As mentioned in Section 4.2, we detect tag tests by observing consecutive reads of a variable where the type is narrowed at the second read. Usually, the first read of this pair will be the use in the type guard. In cases where the guard expression is saved to a local variable before being checked, this technique detected spurious additional guards, but they were easy to recognize and discard manually.

We ran the set of benchmark programs from Section 6.1 with the union-producing $\mathbb{T}.\sqcup$ operator of Section 6.2, and a flow sensitive $\mathbb{T}$`.equiv` policy to keep types of variable occurrences under conditionals separated.

Results. It took 90 minutes to manually classify all of the discovered type guards, the results of which can be seen in Table 2. The figure contains rows for different kinds of type guards, if a type is non-atomic, it can be classified as having multiple kinds. Due to their prevalence, extra rows have been added for type guards that are applied on the property of the guarded value.

Overall there are 6 major classes of narrowing checks: `typeof`, `instanceof`, checking an explicit tag field in a data structure[11] (mostly in the AST manipulations in our corpus), a general predicate function, explicit property presence checks (`hasOwnProperty`), and checks of objects' `prototype` or `constructor` fields. The most prevalent type guards are `typeof`, use of general predicate functions, and (a variety of) property presence tests. Checking an explicit tag field, used to mimic algebraic datatypes in JavaScript, are common but highly program-dependent. Non-local type guards are rare, and would be easy to rewrite if needed. Boolean combination of type guards is common enough that they probably should not be ignored by a sound union elimination check.

6.4 Evaluating Fixed Object Layout

We present a case study of evaluating Choi et al.'s type system [9] to enforce *fixed object layout* in JavaScript, as described in Section 4.4. We checked the following salient features in the trace typing framework:

[11] A popular requested TypeScript feature: `https://github.com/Microsoft/TypeScript/issues/186`

1. *Property writes* We checked that property writes were only performed on properties deemed to be "read-write" properties in the type system.
2. *Precise types* We checked that only objects with a "precise" type (see Section 4.4) were used as a prototype parent for other objects.
3. *Consistent shadowing* We checked that when properties were shadowed in the prototype chain, the parent and child properties had the same type.

We ran trace typing with this type system with the benchmark suite of Section 6.1 (with the subtyping $\mathbb{T}$.`equiv` and `base` policy from Section 6.2, merging all occurrences of a variable in a dynamic function invocation). Table 1 shows the errors reported by our trace typing model. The column *ro/rw* shows, in absolute numbers, how many of the property writes were to fields that were inherited from prototype chain (and not shadowed by a local redefinition), of the total number of property writes. The column *prototypal* shows, how many of the assignments to some object's prototype property was in violation of the restriction mentioned (second bullet) above, vs total number of such assignments. The column *inheritance* shows in how many cases, a property was shadowed with an inconsistent type.

The results add significant confidence to our initial beliefs that some of the restrictions the type system imposes are not onerous in practice: the numerator numbers are generally (though not universally) quite small compare to the denominator. We manually found that most of the *ro/rw* type errors for `lazy.js` were due to the way prototype hierarchies were initialized, and that the type errors could be eliminated by a simple refactoring.

Additionally, from the trace typing data (not presented here) we found that support for *optional properties* in the type system could have helped eliminate a large number of type errors, and so would be a very useful feature to add to the type system. Trace typing finds this information without the need for implementing an actual static type inferencer for JavaScript that is robust enough to work on large code bases.

7 Discussion

In this section, we discuss the quality of types inferred by trace typing, present limitations and threats to validity, and then mention possible future generalizations of our approach.

Quality of Types Inferred. Because our type inference approach is non-standard and our error counts are sensitive to over-generalization (Section 3.5), it is worth asking whether the types we infer are sensible — whether a human developer would write similar types. While there is no systematic way to verify this, our experience says that the types are sensible. In the course of developing the framework and type system plugins, we spent a significant amount of type examining the types inferred. Object (property) types rarely deviated from what we would have written ourselves.

We discuss one specific example of types inferred by trace typing for the `typescript` benchmark, for which the original TypeScript code contains declared types (trace typing analyzes the transpiled JavaScript version). Figure 12 contains an excerpt from the TypeScript code. At the `return` statement the `node` variable always has type `VarLikeDecl` (also shown in the figure). Trace typing finds the relevant properties that we see mentioned in the interface, as well as the inherited properties. Moreover, it also correctly computes the structural type of each property of `VarLikeDecl` such as `initializer`. Due to space reasons, we do not show all the inferred types; but we found them reasonable on manual inspection.

```
// TypeScript/src/compiler/parser.ts:39
export function forEachChild<T>(node:Node, ...):T {
  ... switch (node.kind) {
    ... case SyntaxKind.PropertySignature:
      ... return visitNode(cbNode, (<VarLikeDecl>node).propName);
    }
}
// TypeScript/src/compiler/type.ts:504
export interface VarLikeDecl extends Declaration {
  propName?: Identifier; \n dotDotDotToken?: Node;
  name: DeclarationName; \n questionToken?: Node;
  type?: TypeNode; \n initializer?: Expression;
}
```

Figure 12 Code fragment from `typescript`.

Our ascription of parametric polymorphic types (Section 4.3) also works well in practice. For example, it computes these polymorphic signatures for array methods (from the native environment):

```
Array<E>.prototype.indexOf: (E) -> Number
Array<E>.prototype.concat: (Array<E>) -> Array<E>
Array<E>.prototype.push: (E) -> Number
Array<E>.prototype.pop: () -> E
```

These types are similar to the types used for native environment by TypeScript programs.[12]

The exceptions to the above arise mostly where type errors occur, and in those cases the types are not bizarre but simply a different choice among several incompatible options (e.g., when a method is overridden with an incompatible type). For the **intersect** configuration of Section 6.2, with unbounded intersection types, the gap between inferred types and what one would write at the source level is larger, due to the synthetic nature of the type system. Designing a trace typing system that infers a mix of parametric polymorphism and intersection types closer to what a person would write is future work.

Limitations and Threats to Validity. Trace typing works under the assumption that the input programs are mostly type correct; the error locations generated by our framework lose accuracy when this assumption is violated. Moreover, we assume that most parts of a program *have a correct type* in the system being modeled, which may not always hold. Our manual inspection of inferred types suggests that the natural formulations for $\mathbb{T}.\alpha$ and $\mathbb{T}.\sqcup$ generally behave well, but it is possible other pathological cases exist.

A result that type system design A yields fewer type errors than design B in our framework does *not* strictly imply that the same will hold for complete source-level implementations, due to inevitable over-approximations in a static type checker. Constructs inducing such approximations include reflective constructs like dynamic property accesses (which are resolved in our trace) and loops. Type system feature interactions can exacerbate this effect. The best use of trace typing is to compare type systems that differ in only one or two parameters, and to give less weight to very small differences in the number of errors detected. Consider this example:

```
1 function f(x) { return x.p; }
2 f({p:3, q:4});
3 f({p:3, r:true});
```

[12] https://github.com/Microsoft/TypeScript/blob/v1.7.3/lib/lib.core.d.ts#L1008

```
4 var y = {p:3};
5 y = {p:3, q:5};
6 f(y);
```

Here, type ascription's over-generalizing in the presence of what would be static type errors can lead to some unexpected results. For the example above, ascription in the **base** polymorphism configuration (see Section 6.2) with no union types would produce the type `{p:Number} -> Number` for `f`, which would allow the trace of the program to type check. Ascription with the more precise **intersect** configuration would give the intersection type `{p:Number,q:Number} -> Number` ∧ `{p:Number,r:Boolean} -> Number`. But, the two writes to `y` ensures that it is ascribed type `{p:Number}`, which is not a subtype of either argument type in the intersection! In such cases, a more powerful trace type system can produce more errors than a less powerful one. We did not observe this issue in experimental results we inspected manually, but it may have occurred elsewhere.

Our work is an empirical study using dynamic analysis, and we inherit the normal threats to validity from such an approach, namely possible sampling bias in the subject programs and sampling bias from the inputs used to gather traces. For our study, we took inputs from each program's often-substantial test suite, which appeared from manual inspection to give reasonable path and value coverage.

Potential Extensions. We have focused thus far on type systems for JavaScript, but the core ideas of our technique apply to other languages as well. The generalization to other dynamically typed languages (e.g., Ruby, Python) would require instrumentation of the other language. The type system implementations on top of the new framework would need to be tailored to the primitives and idioms of the new target language.

Our existing trace format could be used for additional experiments that analyze the types ascribed to specific parts of the program. For example, this could inform strategies for typing challenging constructs like computed property accesses [24].

More metadata could be added to our traces to enable experimentation with other type-system features, with no changes to our core approach. For example, more detailed information about the evolving types and layouts of objects could be maintained, to enable experimentation with alias types [27, 10]. Our hope is that trace typing will be a useful base for such future extensions.

8 Related work

We focus here primarily on approaches to retrofitting type systems onto existing languages and on other means of evaluating type systems on large bodies of code, with a secondary focus on type systems for JavaScript.

Retrofitting Type Systems. Retrofitting type systems onto existing languages is not new,[13] nor is it restricted to dynamic languages. The notion of *soft typing* for dynamically typed languages is now well-established [7, 31, 28]. The notion of *pluggable type systems* — including extensions to existing type systems — is also an established idea [6], which continues to produce useful tools [12, 23]. Both bodies of work suffer the same difficulty in evaluating lost precision: it requires tremendous manual effort to evaluate on non-trivial amounts of code, leading to design decisions based on intuition and a handful of carefully-chosen examples.

[13] The use of the term *retrofitting* is relatively recent, due to Lerner et al. [21].

More recently, two groups of researchers arrived at means to induce a gradual type system variant of a given base type system, one as a methodology to produce a gradual type system by careful transformation of a static type system [14] and another as a a tool to compute a gradual version of a type system expressed as a logic program [14]. Both deterministically produce one gradual type system from one static type system, aiding with the introduction of gradual typing, but not of overall design.

Evaluating Type Systems. The implementation effort for prototyping similar but slightly different type systems and applying them is a significant barrier to evaluating completely new type systems for existing programming languages. TeJaS [21] attempts to remedy this for JavaScript type systems by carefully architecting a modular framework for implementing type systems. It relies on bidirectional typing to reduce annotation burden, but this still requires significant manual annotation.

A smaller but closely-related body of work is evaluating changes to existing type systems. Wright studied approximately 250,000 lines of SML code [30], cataloguing the changes necessary to existing code in order to type-check under the value-restriction to type generalization. The study in this case required less implementation effort than the general approach we propose; SML implementations existed, and could be modified to test the single alternative type system change. Nonetheless, his careful analysis of the impact of the proposed change justified a type system change that has withstood the test of time. SML still uses the value restriction, and OCaml uses a mild relaxation of it [15]. More recently Greenman et al. [18] performed a similar evaluation of an alternative proposal for F-bounded polymorphism in Java.

Dynamic Type Inference. Type inference for dynamic languages is another rich area with strong connections to our trace-based type inference. The closest such work to ours is dynamic type inference for Ruby [2]. They use Ruby's reflection capabilities to pass virtualized values through a program, with each use of the virtualized objects gathering constraints on the type of the object. Their system has a guarantee that the inferred types will be sound if every path within each procedure is executed at least once. Like trace typing, they rely on dynamic information to infer types for program fragments, including for complex source-level constructs. Unlike our approach, they make no attempt to simulate the effect of type *checking* the program: they infer types that hold at method entry and exit, but do not infer types for local variables, whose types may change arbitrarily. Also, they cannot *infer* method polymorphism (though they support annotations), require manual annotation for native code, and evaluate only a single type system design.

Saftoiu et al. [25] implement dynamic type inference for JavaScript, generating types for a particular JavaScript type system based on dynamic observation, and find that it is useful for converting code to their typed JavaScript dialect. Their type system is sound with respect to λ_{JS} [19], and they can type-check the source code to verify the inferred types. We cannot do this, because our goal is experimenting with type system design, so our type systems are not adequate to type check source programs. Like us they instrument code via source-to-source translation. We evaluate significantly more code: their examples total 3799 lines of code.

Other Uses of Traces. Coughlin et al. [11] use dynamic traces to identify the spans between program safety checks and data uses guarded by those checks, e.g., a null check followed by a dereference. These measurements can inform the design of static analyses, e.g., by

indicating whether or not an analysis should be interprocedural. While similar in philosophy to our work, we differ substantially in technical details. First, their implementation is specific to null dereference errors. Second, their goal is to *measure* aspects of program behavior quantitatively, and interpret that measurement when designing an analysis; we instead advocate directly comparing approximations of a real analysis.

Åkerblom and Wrigstad [1] study the receiver polymorphism of a very large base of open source Python programs, primarily to examine how adequate nominal subtyping might be for Python (as opposed to structural subtyping). They classify static call sites by whether they were observed to have multiple receiver types, consider whether call sites might be dispatched on parametrically-polymorphic targets by clustering call sites in different ways (roughly similar to our context-sensitivity policies), and do some reasoning about cases where different static calls on the same syntactic receiver may have different qualities (a coarse approximation of typing local variables). They do not reason about actual method types, argument types (except implicitly if methods are invoked on arguments), and do not attempt to simulate source type checking as we do. ECMAScript 5 — the previous version of JavaScript — lacks classes, so even code written to ECMAScript 6 — which added classes — will generally require structural subtyping. Our type ascription machinery could be used in a style similar to Section 6.3 to implement most of their instrumentation.

Type Systems for JavaScript. Most previous work on static typing for JavaScript focuses on novel approaches to typing its most expressive features; we describe particularly relevant examples here. Guha et al.'s flow typing [20] allows for refining types in code guarded by runtime tag checks. DJS [10] implements a sophisticated dependent type system that includes a type-based analogue [27] of separation logic to describe heap changes, and an SMT-based refinement type system for characterizing the shape of objects. TeJaS [21] builds an expressive framework for experimenting with JavaScript type systems, whose core is an extension to $F^{\omega}_{<:}$.

Throughout we have discussed Flow [13] and TypeScript [22, 5]. They are quite similar (objects, unions with unsound elimination rules, intersections) with a few small points of divergence (singleton string types and intentional unsound subtyping in TypeScript, non-nullable fields in Flow). They have very different approaches to type inference and checking: TypeScript uses limited local type inference, while Flow performs a global (per-module) data flow analysis. We have already modeled key features — objects with structural subtyping, union and intersection types — in our framework.

TypeScript and Flow include forms of parametric polymorphism, based on programmer annotations. Trace typing can approximate some forms of parametric polymorphism (Section 3.4). Both TypeScript and Flow also allow some narrowing of union types based on tag tests (unsound in general due to aliasing and mutation). Trace typing could be extended to handle such features in future work, as discussed in Section 7.

Both TypeScript and Flow have added type system features based on user feedback, with great success. Trace typing provides an alternate, complementary source of information for deciding which type system features to include, which can be employed without having a large user base.

9 Conclusions

We presented a framework for quantitatively evaluating variations of a retrofitted type system on large code bases. Our approach involves gathering traces of program executions, inferring

types for instances of variables and expressions occurring in a trace, merging types according to merge strategies that reflect options in the source-level type system design space, and type-checking the trace. By leveraging the simple structure of traces, we dramatically reduce the effort required to get useful feedback on the relative benefit of type system features.

To evaluate the framework, we considered six variations of a type system retrofitted onto JavaScript. In each case, we measured the number of type errors reported for a set of traces gathered from nine JavaScript applications (over 50KLOC) written in a variety of programming styles. The data offers quantitative validation for some of the design choices in two popular retrofitted type systems for JavaScript (Typescript and Flow). In a different experiment, we used the results of trace typing to automatically identify places where type narrowing occurred dynamically, to gather empirical results on the frequency and variety of tag tests in JavaScript. In yet another experiment, we evaluated how onerous the restrictions for a new retrofitted JavaScript type system [9] were, validating intuitions that our restrictions were mostly reasonable, and identifying priorities for future extensions.

The feasibility of carrying out these experiments is a strong validation of the trace typing approach.

References

1 Beatrice Åkerblom and Tobias Wrigstad. Measuring polymorphism in Python programs. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS)*, 2015.

2 Jong-hoon David An, Avik Chaudhuri, Jeffrey S Foster, and Michael Hicks. Dynamic Inference of Static Types for Ruby. In *POPL*, 2011.

3 Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A Framework for Implementing Pluggable Type Systems. In *OOPSLA*, 2006.

4 Esben Andreasen, Colin S. Gordon, Satish Chandra, Manu Sridharan, Frank Tip, and Koushik Sen. Trace Typing: An Approach for Evaluating Retrofitted Type Systems (Extended Version). Technical Report SRA-CSIC-2016-001, Samsung Research America, 2016. URL: `https://arxiv.org/abs/1605.01362`.

5 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *ECOOP*, 2014.

6 Gilad Bracha. Pluggable Type Systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

7 Robert Cartwright and Mike Fagan. Soft Typing. In *PLDI*, 1991.

8 Avik Chaudhuri. Bounded Polymorphism. URL: `http://flowtype.org/blog/2015/03/12/Bounded-Polymorphism.html`.

9 Philip Wontae Choi, Satish Chandra, George Necula, and Koushik Sen. SJS: A Type System for JavaScript with Fixed Object Layout. In *SAS*, 2015.

10 Ravi Chugh, David Herman, and Ranjit Jhala. Dependent Types for JavaScript. In *OOPSLA*, 2012.

11 Devin Coughlin, Bor-Yuh Evan Chang, Amer Diwan, and Jeremy G. Siek. Measuring enforcement windows with symbolic trace interpretation: What well-behaved programs say. In *ISSTA*, 2012.

12 Werner Dietl, Stephanie Dietzel, Michael D Ernst, Kivanç Muşlu, and Todd W Schiller. Building and Using Pluggable Type-Checkers. In *ICSE*, 2011.

13 Facebook. Flow: A Static Type Checker for JavaScript. `http://flowtype.org/`.

14 Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting Gradual Typing. In *POPL*, 2016.

15 Jacques Garrigue. Relaxing the Value Restriction. In *Functional and Logic Programming*, 2004.

16 Google. Closure. https://developers.google.com/closure/.

17 Colin S Gordon, Werner Dietl, Michael D Ernst, and Dan Grossman. JavaUI: Effects for Controlling UI Object Access. In *ECOOP*, 2013.

18 Ben Greenman, Fabian Muehlboeck, and Ross Tate. Getting F-bounded Polymorphism into Shape. In *PLDI*, 2014.

19 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *ECOOP*, 2010.

20 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing Local Control and State Using Flow Analysis. In *ESOP*, 2011.

21 Benjamin S Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. TeJaS: Retrofitting Type Systems for JavaScript. In *DLS*, 2013.

22 Microsoft. TypeScript Handbook. http://www.typescriptlang.org/Handbook.

23 Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical Pluggable Types for Java. In *ISSTA*, 2008.

24 Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Semantics and Types for Objects with First-Class Member Names. In *FOOL*, 2012.

25 Claudiu Saftoiu, Arjun Guha, and Shriram Krishnamurthi. Runtime Type-Discovery for JavaScript. Technical Report CS-10-05, Brown University, 2010.

26 Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *FSE*, 2013.

27 Frederick Smith, David Walker, and Greg Morrisett. Alias Types. In *ESOP*, 2000.

28 Sam Tobin-Hochstadt and Matthias Felleisen. Logical Types for Untyped Languages. In *ICFP*, 2010.

29 Jonathan Turner. Announcing TypeScript 1.4. URL: http://blogs.msdn.com/b/typescript/archive/2015/01/16/announcing-typescript-1-4.aspx.

30 Andrew K. Wright. Simple Imperative Polymorphism. *LISP and Symbolic Computation*, 8(4):343–355, 1995.

31 Andrew K. Wright and Robert Cartwright. A Practical Soft Type System for Scheme. *ACM TOPLAS*, 19(1):87–152, January 1997. doi:10.1145/239912.239917.

32 Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans, and Stephen Tu. The HipHop Compiler for PHP. In *OOPSLA*, 2012.

QL: Object-oriented Queries on Relational Data

Pavel Avgustinov[1], Oege de Moor[2], Michael Peyton Jones[3], and Max Schäfer[4]

1	**Semmle Ltd**	`publications@semmle.com`
2	**Semmle Ltd**	`publications@semmle.com`
3	**Semmle Ltd**	`publications@semmle.com`
4	**Semmle Ltd**	`publications@semmle.com`

Abstract

This paper describes QL, a language for querying complex, potentially recursive data structures. QL compiles to Datalog and runs on a standard relational database, yet it provides familiar-looking object-oriented features such as classes and methods, reinterpreted in logical terms: classes are logical properties describing sets of values, subclassing is implication, and virtual calls are dispatched dynamically by considering the most specific classes containing the receiver. Furthermore, types in QL are prescriptive and actively influence program evaluation rather than just describing it. In combination, these features enable the development of concise queries based on reusable libraries, which are written in a purely declarative style, yet can be efficiently executed even on very large data sets. In particular, we have used QL to implement static analyses for various programming languages, which scale to millions of lines of code.

1998 ACM Subject Classification D.1.5 Object-oriented Programming, D.1.6 Logic Programming, F.3.3 Studies of Program Constructs

Keywords and phrases Object orientation, Datalog, query languages, prescriptive typing

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.2

1 Introduction

QL is a declarative, object-oriented logic programming language for querying complex, potentially recursive data structures encoded in a relational data model. It is a general-purpose query language, but its strong support for recursion and aggregates makes it particularly well suited for implementing static analyses, code queries and software metrics. Although this paper is not about static analysis *per se*, it is in this area that QL, being the technical basis of Semmle's engineering analytics platform, has seen most use so far, so we will use it as our main source of motivating examples.

A static analysis implemented in QL is simply a query run on a special database: the database contains a representation of the program to analyse (encoding, say, its abstract syntax tree or control flow graph), from which the query computes a set of result tuples. A bug finding analysis, for instance, could return pairs of source locations and error messages. Since the database describes the program as it was at one particular point in time, we refer to it as a *snapshot database*. A snapshot database is created by a language-specific *extractor*. We have built extractors for various different languages based on existing compiler frontends.

As our first example of a QL query, let us consider an analysis for finding useless expressions in JavaScript programs, i.e., pure (that is, side effect-free) expressions appearing in a void context where their value is immediately discarded. Typically, this indicates a typo, for instance mistyping an assignment "`x = 42;`" as an equality check "`x == 42;`".

Listing 1 QL query for finding useless expressions in JavaScript.

```
import javascript

predicate inVoidContext(Expr e) {
  exists (ExprStmt s | e = s.getExpr()) or
  exists (SeqExpr seq, int i | e = seq.getOperand(i) and
    (i < count(Expr op | op = seq.getOperand(_))-1 or inVoidContext(seq)))
}

from Expr e
where e.isPure() and inVoidContext(e) and not (e instanceof VoidExpr)
select e, "This expression has no effect."
```

To identify such expressions we need to implement a purity analysis and a check to determine whether an expression appears in a void context. Fortunately, the former is already implemented in our standard QL library for JavaScript, so we can concentrate on the latter.

A simple query for finding useless expressions is shown in Listing 1. At a very high level, it breaks down into three sections:

- An `import` statement pulls in the existing QL library `javascript`, which, as its name suggests, provides general support for working with JavaScript snapshot databases.
- A predicate `inVoidContext` is defined to identify expressions in void context.
- The main `from-where-select` clause defines the analysis itself:
  - the `from` part declares a variable `e` ranging over all expressions in the analysed program;
  - the `where` part imposes three conditions on `e`: it must be pure, appear in a void context, and not be a `void` expression, which explicitly discards the value of its operand;
  - the `select` part specifies the results to return for values of the `from` variables that pass the `where` conditions; in this case, `e` itself is returned with an explanatory message.

Taking a closer look at the definition of `inVoidContext`, it is declared as a unary predicate with a single parameter `e` of type `Expr`. `Expr` and its subtypes model JavaScript expression ASTs: for instance, `BinaryExpr` is a subclass of `Expr` representing all binary expressions, which in turn has a subclass `AddExpr` representing additions; another subclass of `Expr` is `SeqExpr`, representing sequence (or "comma") expressions with two or more operands.

The body of the predicate is a first-order formula with two disjuncts. Its first disjunct says that `e` is in void context if it is the toplevel expression in an expression statement (as in our example above). The second disjunct handles the case where `e` is an operand of a sequence expression: `e` is in void context if it is not the last operand, or if the entire sequence is in void context, as determined by a recursive call to `inVoidContext`.

Judging from this example, QL looks like a domain-specific language for querying JavaScript ASTs, but this not the case: the classes used in this example and the navigation operations available on them are defined entirely in QL, not built into the language. In fact, there is nothing about QL that is specific to dealing with ASTs, or even for writing static analyses, but its object-oriented features allow the development of reusable domain-specific libraries (such as the `javascript` library and its cousins for other languages), providing a rich and convenient API for query writers.

Perhaps more surprisingly, there are not even any objects in the traditional sense of structured records with fields and methods. QL programs only work with atomic values; structured data is encoded as relational tables. For example, it is natural at first to think of the formula `e = seq.getOperand(i)` as an operation on an object `seq`, perhaps involving

reading the i-th element of one of its fields holding an array of references to other objects, and then storing the result in variable `e`. In fact, though, all three variables `e`, `seq` and `i` range over atomic values: the latter is an integer, and the former two are *entity values*, that is, opaque identifiers representing entities modelled in the database (in this case, expressions).

In JavaScript snapshot databases, the expression AST structure of the program is encoded in a relation `exprs` containing 4-tuples (c, k, p, i), for entity values c and p and integers k and i, stating that expression c is the the i-th child of p in the AST, and has kind k. Class `Expr` and its subclasses define an object-oriented view of this relation; for example, `getOperand` is defined such that `e = seq.getOperand(i)` is compiled to $\exists k.$`exprs`$(\texttt{e}, k, \texttt{seq}, \texttt{i})$: no field reads, no assignments, just logic.

In particular, `e` is not an output computed from inputs `seq` and `i`: all three variables are on an equal footing, and there is not even any requirement that `e` is functionally determined by `seq` and `i` (though in this particular case it is). This becomes obvious in our use of the **count** aggregate to determine the number of operands to `seq`: `op = seq.getOperand(_)` holds for any value of `op` that is an operand of `seq` (where "`_`" is the special don't-care variable familiar from other logic languages), so `seq.getOperand(_)` behaves like a multi-valued expression. We use the aggregate to count how many of those values there are to obtain the number of operands of `seq`.

In spite of their unusual semantic underpinnings, QL classes offer very similar features to their more traditional counterparts. In particular, classes can have member predicates, such as the `isPure` predicate on `Expr`, which is defined in the standard QL library for JavaScript and overridden with different implementations for various subclasses of `Expr`. Calls such as `e.isPure()` are dispatched virtually, looking up the most specific applicable definitions of `isPure` based on the (runtime) value of `e`.

Like in Java and many other languages, all variables in QL have statically declared types, offering the usual benefits of enabling smart IDEs.[1] However, type declarations in QL are not just assertions to be checked by the compiler, but do, in fact, affect program semantics at runtime: the values that a variable can take during execution are restricted to those that conform to the declared type. In particular, the declared types of predicate parameters and quantified variables restrict the set of values they may range over.

As mentioned above, our example query makes use of (a small part of) the standard QL libraries for JavaScript. Just like the query, the libraries are implemented in an entirely declarative style, specifying *what* should be computed rather than *how*. In fact, QL exposes no details at all of the underlying database system on which the queries are run, and it is up to the optimiser to translate the high-level QL code into an efficiently executable query plan.

In this paper, we present the core features of QL:

- We explain the semantics of classes, member predicates and virtual dispatch, first informally (Section 2) and then more formally via a translation from a subset of the language, Core QL, to plain Datalog (Section 3).
- We discuss practical usage of QL in Section 4, and report on a case study in using QL to implement static checks for Java in Section 5.
- We put QL into context in Section 6, exploring in detail how far it matches the principles of object orientation laid down in the literature, and briefly survey related work.

[1] In fact, this is the main motivation for choosing the `from-where-select` query syntax instead of SQL's `select-from-where`: variables are declared upfront, so code completion is available in the `select` part.

2 Overview of QL

The fundamental semantic model of QL is that of Datalog: programs define a set of *intensional predicates*, one of which is a distinguished *query predicate*. They are evaluated on top of an *extensional database* (EDB), which defines a set of *extensional predicates*. While intensional predicates are defined by formulas of first-order logic (possibly involving recursion between predicates), extensional predicates are defined as explicit sets of tuples stored in the database. Unlike Prolog, Datalog does not allow the use of complex terms, so intensional predicates can only refer to values already contained in the database and cannot build up new data structures, such as lists. Like many Datalog dialects, QL somewhat relaxes this restriction by providing support for arithmetic and string operations.

The semantics of a program is the least fixpoint of its intensional predicates, that is, intensional predicates are assigned the smallest sets of tuples that satisfy their recursive definitions. Since such a fixpoint need not exist in general, QL imposes the restriction that (mutual) recursion is only allowed under an even number of negations, which is a variant of the *stratified negation* restriction used in many Datalog systems [32]. Once a fixpoint solution has been found, the set of tuples assigned to the query predicate is returned as the overall result of the program.

The grounding of QL's semantics in Datalog is not just an expository device: as explained in Section 4, our implementation compiles QL to plain Datalog, and we shall provide a precise semantics for a core calculus of QL in the next section by formalising the essential parts of that translation.

2.1 Classes

A type in QL represents a set of values, which we will call the *extent* of the type. Classes are types whose extent is defined by a unary intensional predicate called the *characteristic predicate* (or *character* for short) of the class.

There are also two kinds of *base types*, that is, types which are not themselves defined in QL: *primitive types* such as `int` or `string` are built into the language; *entity types* are defined by unary extensional predicates, whose names by convention start with an "@" character. Primitive types always have the same extent, regardless of the content of the EDB, while the extent of entity types and classes may depend on the EDB. For example, snapshot databases representing JavaScript programs defines entity types `@expr` and `@seqexpr` whose extent is, respectively, the set of all expressions and the set of sequence expressions in the represented program.

Subtyping can be thought of as set inclusion of extents: if A is a subtype of B, then the extent of A is a (not necessarily proper) subset of the extent of B.[2] For entity types the subtyping relation is given by the database schema: for instance, the schema for JavaScript snapshot databases declares `@seqexpr` to be a subtype of `@expr`, and it is up to the database system to ensure that this constraint is met at runtime. For classes, direct supertypes are specified as part of their declaration using a Java-like `extends` clause.

While entity types can only be subtypes of other entity types, classes can also extend base types. For instance, we can define a class `Digit` with the extent $\{0, 1, 2, \ldots, 9\}$:

```
class Digit extends int { Digit() { (int)this in [0..9] } }
```

[2] The reverse direction cannot, in general, hold: since characters are arbitrary predicates, inclusion of extents is undecidable, while our subtyping relation needs to be kept decidable.

The `extends` clause makes `Digit` a subtype of the built-in `int` type, and the character (which syntactically looks like a constructor in Java) further restricts the extent of `Digit`. The `x in [a..b]` notation is a convenience for defining ranges (note that an explicit cast is necessary when using variables with a class type in numeric operations).

Characteristic predicates can contain arbitrary QL code. For instance, we can define the class of even digits and the class of prime digits by subclassing `Digit` and performing arithmetic checks on `this`.

```
class Even extends Digit { Even() { (int)this class PrimeDigit extends Digit {
  PrimeDigit() { count(Digit divisor | (int)this
```

Observe that the extents of the two classes overlap, yet neither is a subset of the other. This is a natural consequence of defining types by arbitrary characteristic predicates, but it means that not every value has a unique tightest type.

Like Java, QL has an `instanceof` operator, which in QL is really just syntactic sugar for calling the character of a class. For instance, the class of odd digits can be defined like this:

```
class Odd extends Digit { Odd() { not this instanceof Even } }
```

Being intensional predicates, characters can be recursive. For instance, we could define:

```
class Even extends Digit { Even() { this = 0 or (int)this-1 instanceof Odd } }
class Odd  extends Digit { Odd() { (int)this-1 instanceof Even } }
```

However, recursion has to be stratified: simply defining `Even` and `Odd` as mutual complements is not acceptable, since this definition has no least fixpoint.

A class may extend multiple supertypes, which simply means that it is a subtype of the their intersection. The (potentially trivial) intersection of all supertypes of a class is called the *domain* of the class. For instance, the class of even prime digits ($\{2\}$) is defined as

```
class EvenPrime extends Even, PrimeDigit {}
```

In fact, since `EvenPrime` imposes no additional constraints on `this` in its character, its extent is exactly equal to its domain. In general, the extent of a class consists of all those values in its domain that satisfy the body of the character; hence, the implicit `this` variable in the character ranges over the domain of the class.

It should be emphasised that the constructor-like syntax for characters is purely superficial: QL has no `new` expression. Like plain Datalog, QL programs can never construct new values or objects, they can only work with primitive values and the values present in the EDB.

2.2 Prescriptive typing

Every variable in QL has a declared type. In most statically typed imperative and functional languages, such declarations are purely compile-time artefacts that describe the set of values the variable is allowed to take on at runtime; they are checked for consistency by the compiler but play no role at runtime. In contrast to this *descriptive* typing discipline, QL follows a *prescriptive* model, where the syntactic type declaration corresponds to a semantic containment check at runtime.

For instance, consider the predicate `isSmall` that holds for all `Digit`s smaller than five:

```
predicate isSmall(Digit d) { (int)d < 5 }
```

We can use it in a query like the following (which will return the numbers $0, \ldots, 4$):

```
from int i where isSmall(i) select i
```

Note that `i` is declared to be of type `int`, but is passed as an argument to `isSmall`, whose parameter is declared to be a `Digit`. Under a descriptive typing discipline, this would be a compile-time type error, but not so in QL: declaring `d` to be a `Digit` simply means that in order for a value to satisfy the predicate `isSmall`, it has to *both* be a `Digit` *and* satisfy the logical conditions imposed by the body of the predicate (namely, being smaller than five).

Another way of looking at it is that type declarations entail an implicit `instanceof` test (which is, in fact, made explicit when translating to plain Datalog), and our definition of `isSmall` is equivalent to

```
predicate isSmall(int d) { d instanceof Digit and d < 5 }
```

The call `isSmall(i)` thus has a perfectly well-defined semantics, regardless of the declared type of `i`, and regardless of what set of values `i` ranges over at runtime. If none of these values happen to be in `Digit`, then `isSmall(i)` will evaluate to an empty set of tuples, as in the following query:

```
from int i where isSmall(i) and i < 0 select i
```

In practice, a query or formula that never returns any values usually indicates a mistake. The problem of finding such empty formulas can be reduced to the problem of inferring types for the generated Datalog [33]; (Datalog) formulas for which we infer the empty type are mapped back to the (QL) formulas they arise from, and reported. In general, emptiness of Datalog formulas is undecidable (even without arithmetic or string operations), so we can never find all empty formulas, but in practice this approach has proved to be quite effective.

Besides type declarations, `instanceof` tests and casts also restrict the possible values of variables and expressions: `x instanceof A` restricts `x` to only take on values from `A`, and similarly `(A)x` is an expression picking out those values of `x` that are in `A`.

2.3 Member predicates

The predicate `isSmall` really describes a property of `Digit`s, so it thus makes sense to add it to class `Digit` as a *member predicate*:

```
class Digit extends int {
  Digit() { (int)this in [0..9] }
  predicate isSmall() { (int)this < 5 } }
```

Like characteristic predicates, member predicates have an implicit parameter `this`, and they are invoked using a method call-like syntax:

```
from Digit d where d.isSmall() select d
```

Of course, member predicates can have other parameters besides `this`. For instance, we could add a predicate `divides` to check whether one digit is a divisor of another:

```
class Digit extends int {
  ...
  predicate divides(Digit that) { (int)that
```

There is one important difference between characters and member predicates: in the former, `this` ranges over the domain of the class (that is, the intersection of the extents of its supertypes), while in the latter `this` ranges over the extent of the class itself. This is because the character is what defines the extent of the class in the first place, so by restricting `this` to range over the extent of the class in the character, we would introduce a direct recursive call from the character to itself, which under least fixpoint semantics would mean that the character (and hence the extent of the class) is always empty.

2.4 Multi-valued expressions

Taking the analogy between member predicates and methods further, QL allows treating predicates as multi-valued "functions" with a dedicated `result` parameter. For instance, the member predicate `divides` could instead be written as a multi-valued function returning any of the divisors of a digit:

```
Digit getADivisor() { (int)this
```

Note that member predicates using the function syntax have an implicitly declared `result` variable whose type is the declared result type. The results of the predicate are precisely those values that the `result` variable is bound to. Syntactically, calls to predicates in function syntax are treated like function calls; in particular, they can be chained as in `d.getADivisor().getADivisor()`, which evaluates to all divisors of divisors of `d`.

Semantically, however, such predicates are still relations: there is no requirement that `result` has precisely one value for each value of `this`, or that `result` is "computed from" `this` in some operational sense. In fact, it is quite possible to use `getADivisor()` in reverse to compute all values of `this` yielding a given `result` value, as shown in the following query:

```
from Digit d where d.getADivisor() = 2 select d // selects 0, 2, 4, 6, 8
```

When translating to Datalog, predicates using the function syntax are desugared into normal predicates by making the `result` parameter explicit and introducing temporary variables as necessary. For instance, `d.getADivisor()=2` is translated into code of this form:

```
exists (Digit tmp | d.getADivisor(tmp) and tmp = 2)
```

Thus, multi-valued expressions are a purely syntactic, if practically very useful, feature.

2.5 Overriding and virtual dispatch

Given that we have classes that contain member predicates and that may extend each other, it is natural to ask whether there is a notion of overriding and virtual dispatch, and indeed there is: intuitively, at runtime a call `x.p(...)` is dispatched to the definition of `p` belonging to the tightest class containing `x`, i.e., the most specific applicable definition of `p`.

There are two sources of ambiguity: first, `x` may, in general, have multiple values; this is solved by dispatching the call separately for each individual value. Second, classes may overlap, so even for a single value of `x` there can be multiple most specific definitions of `p`; this is solved by dispatching to each definition separately and unioning the results.

More formally, let us represent member predicates by *relation specifiers* of the form $C.p/n$, where C is the name of the class in which the predicate is declared, p is the name of the predicate itself, and n is the predicate's arity, not including the result parameter.[3] We say that a predicate $C.p/n$ *overrides* a predicate $C'.p/n$ if C is a transitive subtype of C'; in this case, we also say that $C.p/n$ is *more specific* than $C'.p/n$.

A member predicate is a *root definition* (or *rootdef* for short) if it does not override any other predicate. The set of rootdefs of a predicate is the set of all rootdefs that it overrides, or the predicate itself if it is already a rootdef. Note that due to multiple inheritance a predicate can have more than one rootdef, but every predicate has at least one.

[3] QL allows overloading, so there may in fact be multiple member predicates with the same arity as long as they have different parameter types. Like in Java, overloading is resolved entirely statically based on declared types, and hence plays no role in virtual dispatch, and we shall ignore it for simplicity.

The *static target* of a member predicate call `x.p(...)`, where the declared type of `x` is a class C, is the most specific predicate $D.p/n$ such that C is a reflexive, transitive subtype of D and n is the number of arguments in the call. In a valid QL program, every predicate call must have a unique static target. The *dispatch candidates* of `x.p(...)` are all the rootdefs of the static target, as well as any predicates that override at least one of the rootdefs.

At runtime, for every value v of `x`, the *applicable targets* of the call are those dispatch candidates $D.p/n$ for which v is in the extent of D, and the *actual targets* are the most specific applicable targets. The call is dispatched to all actual targets for each value of `x`.

In summary, dispatch for a call `x.p(...)` occurs in two stages, one static and one dynamic. At compile-time we compute the set of dispatch candidates, which contains all rootdefs of `p` above the declared type of `x` (that is, member predicates with the same signature that do not themselves override another definition) and all methods that override them. At runtime, each of these candidates applies only if the value of `x` is contained in the corresponding class, and there is no more specific class that also contains `x`.

For example, assume we add a member predicate `kind` to class `Digit` like this:

```
class Digit extends int { ... string kind() { result = "digit" } }
```

We override `kind` in the subclasses of `Digit` to result in `"even"` for `Even`, `"odd"` for `Odd` and `"even prime"` for `EvenPrime`. Now consider this query:

```
from Even e select e, e.kind()
```

The static target of the call `e.kind()` is `Even.kind/0`, whose (unique) root definition is `Digit.kind/0`. The dispatch candidates are `Digit.kind/0`, `Even.kind/0`, `Odd.kind/0` and `EvenPrime.kind/0`. Since `e` is declared to be an `Even`, it ranges over the set $\{0, 2, 4, 6, 8\}$. For the runtime values 0, 4, 6 and 8, the applicable targets of `e.kind()` are `Digit.kind/0` and `Even.kind/0`, and the (unique) actual target is `Even.kind/0`. For the value 2, the applicable targets are `Digit.kind/0`, `Even.kind/0` and `EvenPrime.kind/0`, and the actual target is `EvenPrime.kind/0`. Hence, the query evaluates to {(0, `"even"`), (2, `"even prime"`), (4, `"even"`), (6, `"even"`), (8, `"even"`)}.

Now consider what happens if we add a new class

```
class Two extends Digit { Two() { this = 2 } string kind() { result = "two" } }
```

`Two.kind/0` has `Digit.kind/0` as its root definition, so it is now also a dispatch candidate for `e.kind()`, and it is an applicable target for $e = 2$. We now have *two* applicable targets in this case, neither of which is more specific than the other. Hence they will *both* be called, so the query additionally returns the tuple `(2, "two")`.

If, on the other hand, we define `Two` to extend `int` instead of `Digit`, its extent does not change, but `Digit.kind/0` is no longer a root definition of `Two.kind/0`, which hence is no longer a dispatch candidate for `e.kind()`.

Discussion

Support for multiple actual targets is perhaps the most unusual feature of virtual dispatch in QL and can be confusing to novice QL programmers. Its main motivations are:

Naturality Virtual dispatch selects the most specific implementation; with overlapping classes there may be more than one, so in a logic language it is natural to take their disjunction.

Simplicity Outlawing multiple actual targets is difficult, since it is undecidable whether two classes overlap (by undecidability of emptiness in Datalog with negation). Requiring

characteristic predicates to be decidable would rule out many practically interesting cases, while checking ambiguity at runtime and aborting with an error seems undesirable.

Usefulness Some common idioms use this feature. For example, a flow analysis could be implemented as a member predicate on data flow nodes; different overriding definitions handle different kinds of flow, and hence naturally overlap. For instance, one definition could model intra-procedural def-use chains, while another models inter-procedural argument passing; both definitions overlap for parameters that are reassigned.

In particular, the last example above rules out a simpler approach where each predicate is considered its own rootdef, which would make dispatch depend very strongly on static types. As another extreme, one could consider all predicates of the right name and arity as dispatch candidates, ignoring static types altogether. This seems undesirable in practice, since it can cause dispatch to completely unrelated predicates that just happen to have the same name. Instead, QL views a rootdef and all its overriding methods as alternative implementations of the same operation. For a given call, all operations implemented by the static target are determined at compile time, and at runtime the most specific implementations are selected.

There is no intrinsic connection between multiple call targets and multiple inheritance: the former arises without the latter, for example if a class defines a predicate that is overridden by two overlapping subclasses. Ambiguous inheritance is, in fact, illegal in QL (as we shall discuss below), and hence does not give rise to multiple call targets.

2.6 Abstract classes

QL classes as we have described them so far lend themselves quite well to top-down modelling: starting from a general superclass representing a large set of values, we carve out individual subclasses representing more restricted sets of values. In particular, the extent of a class is always defined by filtering its domain through the body of its character.

A classic example where this approach is useful is when modelling ASTs: the node types of an AST form a natural inheritance hierarchy, where, for example, there is a class `Expr` representing all expression nodes, with many different subclasses for different categories of expressions; there might, for instance, be a class `ArithmeticExpr` representing arithmetic expressions, which in turn could have classes `AddExpr` and `SubExpr`.

In other cases, however, we might prefer to instead think of a class as being the union of its subclasses. Here, the superclass exists purely as an interface that provides certain member predicates, with subclasses filling in concrete implementations.

QL supports a notion of abstract classes that allow us to do exactly this: like a concrete class, an abstract class has one or more superclasses and a characteristic predicate. However, the extent of an abstract class is not the set of values that satisfies its character, but rather the union of the extents of all its subclasses. In particular, an abstract class without subclasses has an empty extent. We will present a practical example of an abstract class in Section 4.

2.7 Miscellanea

QL has various other language features that are important in practice but are either not semantically fundamental, or have direct counterparts in other Datalog dialects.

As we mentioned at the beginning of this section, QL predicates may be recursive, and our program analysis queries make heavy use of this feature. A particularly common kind of recursion is transitive closure, for which QL offers a syntactic shorthand: for a binary predicate `p`, `p+` denotes its transitive closure, and `p*` its reflexive transitive closure. Obviously, this is purely syntactic sugar that is easily translated into plain recursion.

$prog$	::=	$\overline{cd}\ \overline{pd}$	program
cd	::=	$\textbf{abstract}^?\ \textbf{class}\ C\ \textbf{extends}\ \overline{T}\ \{C()\ \{f\}\ \overline{pd}\}$	class definition
pd	::=	$\textbf{predicate}\ p(\overline{T\ x})\ \{f\}$	predicate definition
f, g	::=	$p(\overline{x}) \mid x.p(\overline{y}) \mid C.\textbf{super}.p(\overline{x}) \mid \textbf{not}\ f$	formula
	|	$f\ \textbf{and}\ g \mid f\ \textbf{or}\ g \mid \textbf{exists}(T\ x \mid f)$	
S, T	::=	$C \mid @b \mid C.\texttt{domain}$	type reference

Figure 1 Syntax of Core QL; $\bar{\cdot}$ denotes (possibly empty) sequences, $\cdot^?$ optional elements.

In addition to virtual calls, QL also provides statically dispatched `super` calls of the form `C.super.p(...)`. Their static target is looked up in `C` (which must be a superclass of the enclosing class), and serves as the single actual target.

The `import` statement makes definitions from one module available in another. For instance, the QL standard library for JavaScript is split into 40 modules, which are all imported into a single module `javascript.qll`. Since imports are transitive, queries can simply import that module to gain access to the entire library (cf. Listing 1). As in Java, implementation hiding is facilitated by access modifiers: member predicates may be marked `private`, meaning that they cannot be called from outside the enclosing class.

QL supports aggregates to perform arithmetic operations such as sum or average on (multi-)sets of values. While very useful in practice, aggregates are really a feature of the Datalog dialect into which QL is compiled, and they do not interact with the language's object-oriented features, hence we will not further discuss them.

As a syntactic convenience, casts may be written in a postfix form as `x.(A)`, which is semantically equivalent to `(A)x`, but saves parentheses in chained calls.

Finally, member predicates of abstract classes may themselves be abstract, meaning that they do not have a body, and the QL compiler checks that each subclass provides an overriding definition of the predicate. Observe that our definition of virtual dispatch guarantees that an abstract member predicate is never the actual target of a call: since the extent of the abstract class is the union of the extents of its subclasses and since each of those subclasses overrides the abstract predicate, there must always be at least one more specific applicable target. Thus, abstract predicates are not semantically fundamental, and in particular have no deep semantic connection with abstract classes.

3 Semantics of Core QL

To formally describe the semantics of QL, we concentrate on a subset dubbed *Core QL* that captures the object-oriented features of QL, while omitting other features that are either purely syntactic or are semantically orthogonal. The semantics of Core QL will be described by a translation to plain Datalog.

3.1 Core QL

Figure 1 presents the syntax of Core QL. Like full QL, Core QL programs consist of toplevel predicates and (concrete and abstract) classes with a characteristic predicate and member predicates. We do not model QL's `from-where-select` query syntax, but simply consider queries as special toplevel predicates.

Predicates can declare parameters, and their bodies are first-order formulas with predicate calls as atomic formulas. As in full QL, there are calls to toplevel predicates and to member predicates, and the latter may be either virtual calls or **super** calls. Unlike full QL, **super** calls always have to be explicitly annotated with the class they refer to.

Type references appearing in **extends** clauses, parameter declarations or existential quantifiers are either class names C or base type names $@b$. We assume the latter to be defined by an underlying database schema. Core QL also has a syntax for *domain types* of the form C.`domain` for a class name C; these cannot appear at the source level but play a crucial role in the semantics of classes.

Among the QL features omitted from Core QL are overloading, the function syntax for predicates, expressions, the **forall** quantifier, casts and **instanceof**: these can all be desugared into Core QL features. Other QL features such as primitive types and aggregates have no counterpart in Core QL, but their semantics is largely orthogonal to the object-oriented features of the language, which are the focus of our presentation.

3.2 Datalog

The target language for our translation is an untyped variant of Datalog. A Datalog program consists of a series of intensional predicate definitions of the form $p(\overline{x}) \leftarrow \varphi.$, where p is a predicate name, $\overline{x}$ is a possibly empty sequence of variable names, and φ is a formula of first-order logic with the usual logical connectives. The free variables of φ must be exactly $\overline{x}$. The atoms of φ are calls of the form $r(\overline{y})$, where r is either the name of an intensional predicate defined in the same program, or the name of an extensional predicate.

We say that an intensional predicate p calls a predicate q, written $p \rightarrow q$, if the body of p contains a call to q. As usual, $\rightarrow^*$ denotes the reflexive transitive closure of this relation. $p \stackrel{-}{\rightarrow} q$ means that one of the calls to q in p occurs under an odd number of negations.

We require all Datalog programs to be *stratified*, that is, recursive call chains of the form $p \rightarrow^* q \stackrel{-}{\rightarrow} r \rightarrow^* p$ are not allowed. Any stratified Datalog program has a least fixpoint semantics, that is, given an interpretation of the extensional predicates, each intensional predicate has a unique minimal interpretation that satisfies the predicate's definition.

3.3 Valid Core QL

In order to be meaningfully translatable to Datalog, a Core QL program has to fulfil a set of syntactic requirements and pass some static semantic checks. There is one additional check that is easiest to perform on the generated Datalog and will be discussed later.

The syntactic requirements are entirely standard and mostly naming related:

▶ **Definition 1** (Syntactic validity)**.** In order for a Core QL program to be *syntactically valid*, the following conditions have to be satisfied:

- No two classes and no two toplevel predicates with the same arity may have the same name; no two member predicates of the same class with the same arity, and no two parameters of the same predicate may have the same name.
- Every **extends** clause must list at least one type.
- Every characteristic predicate must have the same name as its enclosing class.
- No predicate parameter may have the name **this**.
- For every variable name appearing in a formula, there must either be an enclosing **exists** declaring a variable of that name, or the enclosing predicate must have a parameter of that name, or the variable name is **this** and it appears in a member predicate or character. In particular, every variable name can be associated with a declared type.

- Similarly, for every class name appearing in a type reference there must be a class of the same name, and for every predicate name appearing in a call to a toplevel predicate, there must be a toplevel predicate of that name with the appropriate arity.
- **super** calls may only appear in member predicates.

To formulate the static semantic checks, we first introduce some terminology.

▶ **Definition 2** (Relation specifiers)**.** A *relation specifier* $C.p/n$ consists of a class name C and a pair p/n, where p is a predicate name and n a natural number.

Unless otherwise specified, we require relation specifiers to be *valid*, that is, C must be the name of a class defined in the program, and C must declare a member predicate p with n parameters. We abbreviate $C.p/n$ as $C.p$ where n is not important or obvious from context.

▶ **Definition 3** (Subtyping)**.** The *subtyping* relation $S <: T$ is the smallest relation such that for every class C we have $C <: C.\texttt{domain}$, and if C extends T, then $C.\texttt{domain} <: T$.

As usual, $S <:^+ T$ denotes the transitive closure of this relation.

▶ **Definition 4** (Overriding)**.** $C.p/n$ overrides $D.p/n$, written $C.p/n \prec D.p/n$, if $C <:^+ D$. We write $C.p/n \preceq D.p/n$ to mean that either $C = D$ or $C.p/n \prec D.p/n$. If $D.p/n$ overrides no other member relation, it is a *rootdef*. We write $\rho(C.p/n)$ for the set of all rootdefs $D.p/n$ such that $C.p/n \preceq D.p/n$.

▶ **Definition 5** (Member predicate lookup)**.** We define a lookup function $\lambda(S, p, n)$ that looks up a member predicate in a type given a name and its arity and returns a set of candidates:

$$\lambda(S,p,n) = \begin{cases} \{C.p/n\} & \text{if } S = C \text{ and } C.p/n \text{ is valid} \\ \bigcup_{S<:T} \lambda(T,p,n) & \text{otherwise} \end{cases}$$

The static semantic checks guarantee that a program can be translated to Datalog:

▶ **Definition 6** (Translatability)**.** A syntactically valid Core QL program is *translatable* if the following conditions are met:

- It is not the case that $T <:^+ T$ for some type T; that is, the subtyping relation is acyclic.
- For every (not necessarily valid) relation specifier $C.p/n$, we have $|\lambda(C,p,n)| \leq 1$; in other words, classes must override ambiguously inherited predicates.
- For every member predicate call $x.p(\overline{y})$ where x has type T we have $\lambda(T,p,|\overline{y}|) \neq \emptyset$, i.e., all calls can be resolved to a static target.
- Similarly, for every call $D.\textbf{super}.p(\overline{x})$ in a member predicate of a class C, we must have $C <:^+ D$ and $\lambda(D,p,|\overline{x}|) \neq \emptyset$.

3.4 Translation to Datalog

The translation from (translatable) Core QL to Datalog is presented in Figure 2 as a family of structurally recursive translation functions:

- $\mathcal{T}_c$ translates Core QL class definitions into sequences of Datalog predicates, using an auxiliary function $\mathcal{K}$ to generate the extent predicate as explained below;
- $\mathcal{T}_m$ translates Core QL member predicates into Datalog predicates; it takes the declaring class of the member predicate as an additional argument;
- $\mathcal{T}_p$ translates toplevel Core QL predicates into Datalog predicates;
- $\mathcal{T}_b$ translates Core QL predicate and character bodies into Datalog formulas; it takes a type environment as an additional argument;

Translation of a class definition $cd \equiv \textbf{abstract}^? \textbf{ class } C \textbf{ extends } \overline{T}\ \{C()\ \{f\}\ \overline{pd}\}$:

$$\mathcal{T}_c(cd) := \begin{array}{lll} C.\texttt{domain}(\textbf{this}) & \leftarrow & \bigwedge_{C<:B} B.B(\textbf{this}) \wedge \bigwedge_{C<:@b} @b(\textbf{this}). \\ C.C(\textbf{this}) & \leftarrow & \mathcal{T}_b(f, \langle \textbf{this} := C.\texttt{domain} \rangle). \\ C(\textbf{this}) & \leftarrow & \mathcal{K}(cd). \\ \overline{\mathcal{T}_m(pd_i, C)} & & \end{array}$$

$$\mathcal{K}(cd) := \bigvee_{D<:C} D(\textbf{this}) \quad \text{if } cd \text{ is abstract}$$

$$\mathcal{K}(cd) := C.C(\textbf{this}) \quad \text{if } cd \text{ is concrete}$$

Translation of a toplevel predicate definition $pd \equiv \textbf{predicate}\ p(\overline{T\ x})\ \{f\}$:

$$\mathcal{T}_p(pd) := p(\overline{x}) \leftarrow \mathcal{T}_b(f, \langle \overline{x_i := T_i} \rangle).$$

Translation of a member predicate definition $pd \equiv \textbf{predicate}\ p(\overline{T\ x})\ \{f\}$:

$$\mathcal{T}_m(pd, C) := \begin{array}{lll} C.p(\textbf{this}, \overline{x}) & \leftarrow & \mathcal{T}_b(f, \langle \textbf{this} := C, \overline{x_i := T_i} \rangle). \\ C.p^{\text{disp}}(\textbf{this}, \overline{x}) & \leftarrow & \left(\bigwedge_{D.p \prec C.p} \neg D(\textbf{this}) \right) \wedge C.p(\textbf{this}, \overline{x}). \end{array}$$

Translation of a predicate or character body f:

$$\mathcal{T}_b(f, \Gamma) := \left(\bigwedge_{(x,S)\in\Gamma} S(x) \right) \wedge \mathcal{T}_f(f, \Gamma)$$

Translation of a predicate call:

$$\begin{array}{lll} \mathcal{T}_f(p(\overline{x}), \Gamma) & := & p(\overline{x}) \\ \mathcal{T}_f(x.p(\overline{y}), \Gamma) & := & \bigvee_{R.p\in\rho(D.p)} \left(\bigvee_{B.p \preceq^* R.p} B.p^{\text{disp}}(x, \overline{y}) \right) \quad \text{where } D.p := \lambda(\Gamma(x), p, |\overline{y}|) \\ \mathcal{T}_f(C.\textbf{super}.p(\overline{x}), \Gamma) & := & D.p(\textbf{this}, \overline{x}) \quad \text{where } D.p := \lambda(C, p, |\overline{x}|) \end{array}$$

Translation of other formulas:

$$\begin{array}{lll} \mathcal{T}_f(\textbf{not } f, \Gamma) & := & \neg \mathcal{T}_f(f, \Gamma) \\ \mathcal{T}_f(f \textbf{ and } g, \Gamma) & := & \mathcal{T}_f(f, \Gamma) \wedge \mathcal{T}_f(g, \Gamma) \\ \mathcal{T}_f(f \textbf{ or } g, \Gamma) & := & \mathcal{T}_f(f, \Gamma) \vee \mathcal{T}_f(g, \Gamma) \\ \mathcal{T}_f(\textbf{exists}(C\ x \mid f), \Gamma) & := & \exists x.\, (C(x) \wedge \mathcal{T}_f(f, \Gamma[x := C])) \end{array}$$

Figure 2 Translation from Core QL to Datalog (for readability, we write $C <: T$ to mean $C.\texttt{domain} <: T$).

- $\mathcal{T}_f$ translates Core QL formulas into Datalog formulas; it takes a type environment as an additional argument.

The type environments Γ used by $\mathcal{T}_b$ and $\mathcal{T}_f$ are partial functions from variable names to type references. We write $\langle x := T \rangle$ to denote the type environment that maps x to T, and contains no other mappings. As usual, $\Gamma[x := T]$ is a type environment that is identical to Γ except that it maps x to T. $\bigvee_{i \in I} \varphi_i$ and $\bigwedge_{i \in I} \varphi_i$ denote disjunctions and conjunctions of families of formulas indexed by a set I. For empty index sets we define $\bigvee_{i \in \emptyset} \varphi_i := \bot$ and $\bigwedge_{i \in \emptyset} \varphi_i := \top$, i.e., empty disjunctions are false and empty conjunctions are true.

We now discuss the individual translation functions in greater detail.

Classes

For every Core QL class C, we generate a definition for its domain predicate $C.\texttt{domain}$, its characteristic predicate $C.C$, and its extent predicate C. Additionally, each member predicate pd_i is translated recursively using $\mathcal{T}_m$.

The domain predicate is the intersection of the characteristic predicates of all supertypes of C. The characteristic predicate is generated from its Core QL definition, additionally enforcing prescriptive typing, which is not a feature of plain Datalog. The extent predicate, finally, is the Datalog predicate that actually defines the extent of the class. For concrete classes, this is the same as the characteristic predicate. For abstract classes, however, their extent is instead defined as the union of the extents of their subclasses.

The distinction between these three predicates is subtle, but crucial. $C.\texttt{domain}$ is mainly needed to circumscribe the type of **this** inside the characteristic predicate of C. To see why it cannot have type C, consider what the definitions of the characteristic predicate and the extent predicate would look like if it did:

$$\begin{array}{lcl} C.C(\textbf{this}) & \leftarrow & C(\textbf{this}) \wedge \ldots. \\ C(\textbf{this}) & \leftarrow & C.C(\textbf{this}). \end{array}$$

Note the recursion between $C.C$ and C, which is resolved by computing least fixpoints. Clearly, both rules are satisfied if $C.C$ and C are empty, and this is also the least fixpoint. In other words, typing **this** as C in the character would render every concrete class empty. Using type $C.\texttt{domain}$ instead breaks the recursion, and both predicates are now interpreted as the subset of the extent of $C.\texttt{domain}$ that satisfies the body of the character, as expected.

The distinction between the characteristic predicate $C.C$ and the extent predicate C is only relevant for abstract classes (and the two are indeed equal for concrete classes): the extent of an abstract class is not the extent of its characteristic predicate, but rather the union of the extents of its subclasses, and this is precisely how C is defined.

Predicates

Toplevel Core QL predicates are translated directly into Datalog predicates of the same name, using $\mathcal{T}_b$ to translate the body and enforce prescriptive typing for all parameters.

Member predicates $C.p$ are translated into two Datalog predicates: an implementation predicate of the same name, and a dispatch predicate $C.p^{\text{disp}}$. The latter is used during dispatch translation as explained below.

Formulas

Most Core QL formulas are straightforward to translate into their Datalog counterparts, except that for quantifiers we again enforce prescriptive typing. The two most interesting

cases are **super** calls and virtual calls. For the former, we simply use the λ function to look up the member predicate to invoke, explicitly passing in **this** as the first argument.

For virtual calls, we need to implement dispatch. Recall that for a call with static target $D.p$, the dispatch candidates are all member predicates that override a rootdef for $D.p$. Hence the call is translated into two nested disjunctions: the outer disjunction is over all rootdefs $R.p$ of the static target $D.p$, while the inner is over all methods overriding the rootdef. For each dispatch candidate $B.p$ identified in this way, we emit a call to $B.p^{disp}$, which in turn invokes $B.p$, but only if it is a most specific implementation of p for the given parameter.

The syntactic and static semantic checks ensure that the result of the translation is a valid Datalog program, except that they do not yet ensure stratification. While it would be possible to devise a QL-level check for this, it is conceptually simpler to check stratification of the generated Datalog, and map any violations of this condition back to the QL code it originated from. This is also how we implement the stratification check in our QL compiler.

3.5 Example

As a concrete example of the translation, we show the Datalog definitions generated for classes `Digit` and `Even` from Section 2, including definitions for the method `kind` and a query predicate that computes all even digits `e` and their kinds `k`.[4]

$$\texttt{Digit.domain}(\texttt{this}) \leftarrow \texttt{int}(\texttt{this}).$$
$$\texttt{Digit.Digit}(\texttt{this}) \leftarrow \texttt{Digit.domain}(\texttt{this}) \wedge \texttt{range}(\texttt{this}, 0, 9).$$
$$\texttt{Digit}(\texttt{this}) \leftarrow \texttt{Digit.Digit}(\texttt{this}).$$
$$\texttt{Digit.kind}(\texttt{this}, \texttt{result}) \leftarrow \texttt{Digit}(\texttt{this}) \wedge \texttt{string}(\texttt{result}) \wedge \texttt{result} = \texttt{"digit"}.$$
$$\texttt{Digit.kind}^{\text{disp}}(\texttt{this}, \texttt{result}) \leftarrow \neg\texttt{Even}(\texttt{this}) \wedge \texttt{Digit.kind}(\texttt{this}, \texttt{result}).$$
$$\texttt{Even.domain}(\texttt{this}) \leftarrow \texttt{Digit.Digit}(\texttt{this}).$$
$$\texttt{Even.Even}(\texttt{this}) \leftarrow \texttt{Even.domain}(\texttt{this}) \wedge \texttt{mod}(\texttt{this}, 2, 0).$$
$$\texttt{Even}(\texttt{this}) \leftarrow \texttt{Even.Even}(\texttt{this}).$$
$$\texttt{Even.kind}(\texttt{this}, \texttt{result}) \leftarrow \texttt{Even}(\texttt{this}) \wedge \texttt{string}(\texttt{result}) \wedge \texttt{result} = \texttt{"even"}.$$
$$\texttt{Even.kind}^{\text{disp}}(\texttt{this}, \texttt{result}) \leftarrow \texttt{Even.kind}(\texttt{this}, \texttt{result}).$$
$$\texttt{query}(\texttt{e}, \texttt{k}) \leftarrow \texttt{Even}(\texttt{e}) \wedge \texttt{string}(\texttt{k}) \wedge (\texttt{Digit.kind}^{\text{disp}}(\texttt{e}, \texttt{k}) \vee \texttt{Even.kind}^{\text{disp}}(\texttt{e}, \texttt{k})).$$

4 QL in Practice

The previous two sections have presented the semantics of QL in some detail, using toy examples for simplicity. We now show how these concepts apply in a more realistic setting.

4.1 Databases and schemata

Recall that QL programs (or rather, the Datalog programs into which they are translated) are run on a relational database. In practice, we use our own custom database system, but in principle QL programs could just as well be run on an off-the-shelf system.[5]

As in any relational database, data is represented in terms of tuples (rows), grouped into relations (tables) such that all tuples in a relation have the same arity (number of columns).

[4] Core QL does not include primitive types or arithmetic operations, so for the purposes of this example we have treated `int` and `string` like entity types, and assumed EDB relations `range(a,b,c)` and `mod(x,y,z)` corresponding to QL's range operator `a in [b..c]` and the modulo operator `x%y=z`, respectively.

[5] In fact, early versions of our compiler translated QL to SQL, using a third-party database system as our backend. Performance was disappointing, since typical QL compiles to very complex SQL with deeply nested joins and lots of recursion. Most SQL systems are not designed with such queries in mind, and hence handle them badly. On the other hand, our queries do not make use of complex extra-logical features such as database updates or transactions, which made it relatively easy to implement our own engine and saves us some performance overhead.

A table may have a distinguished *primary key* column, meaning that no two rows in the table have the same value in this column; in other words, each value in the primary key column uniquely identifies a row. A table t_1 may also have *foreign key* columns referring to the primary key column of a table t_2 ($t_1 = t_2$ is allowed), meaning that any value occurring in the foreign key column must also be present in the corresponding primary key column; in other words, each row of t_1 references a unique row of t_2.

Keys can be used to model hierarchical data structures using flat tables. Suppose, for instance, that we want to build a snapshot database representing a JavaScript program. Among other data, we may want to represent the abstract syntax tree of source files, including, in particular, all expressions. Simplifying somewhat, we could introduce a table `exprs` with four columns: a primary key column `id` with a unique ID for each expression; a column `kind` indicating what kind of expression we are dealing with, encoded as an integer; a foreign key column `parent` referencing the ID of the parent expression in the AST; and an integer column `idx` recording the ordering among children of the same parent.[6] Since `id` is a primary key, every expression is guaranteed to have a unique ID, and since `parent` is a foreign key, it is a well-defined reference to another expression in the same table.

For example, assume we want to represent a comparison expression `x === 1`; its two children are the variable reference `x` and the integer literal `1`. Assume further that we assign them the IDs 0, 1 and 2, and encode "equality expression" as kind 2, "variable reference" as kind 1, and "integer literal" as kind 0. The variable reference `x` thus has `id` 1, `kind` 1, `parent` 0, and `idx` 0, corresponding to the tuple `exprs(1, 1, 0, 0)`, while "`1`" gives rise to `exprs(2, 0, 0, 1)`. In practice, we would additionally store the name of the referenced variable and the value of the integer literal in separate tables, which we elide for simplicity.

At the storage level, all four columns of the `exprs` table look the same: they are just integers. QL, on the other hand, espouses a strongly typed view where keys are treated as opaque values and annotated with an entity type. Primary key columns define an entity type whose extent is the set of values occurring in that column. For instance, the `id` column of `exprs` could be annotated with the entity type `@expr`, meaning that it defines the extent of entity type `@expr`. Foreign key columns are also annotated with entity types, and the database system ensures that they only contain values drawn from the extent of their type.

This information about tables, the types of their columns, and the entity types they define is described by a *schema*, which thus defines the interface between a QL program and the database on which it is run.

4.2 Data abstraction

Given a snapshot database with a schema, we could write our analyses in plain Datalog, directly accessing the information stored in the tables. However, this can become quite cumbersome since we need to remember which column contains which piece of information, and is not robust against schema changes.

QL classes hide the specifics of how data is stored in tables behind a higher-level interface, thereby acting like abstract datatypes. For instance, we could implement a QL class `Expr` to provide an abstract view of the `exprs` table discussed above:

```
class Expr extends @expr {
  Expr getParent() { exprs(this, _, result, _) }
```

[6] Alternatively, each expression could keep references to their child expressions, but as different kinds of expressions have different numbers of children, this would require tuples with different arities, which would have to be stored in different tables.

```
  Expr getChildExpr(int i) { exprs(result, _, this, i) }
  string toString() { result = "expr" } }
```

Since `Expr` should contain *all* expressions represented in the database, it has a trivial character, and hence the same extent as `@expr`. The member predicate `getParent` provides access to the `parent` column of the `exprs` table, while `getChildExpr` enables navigation in the other direction. Note that we do not need to check that the index `i` is in range: if there is no i-th child, the predicate will simply fail to hold. QL also requires each class to define (or inherit) a `toString` member predicate, for which we provide a dummy implementation.

This interface allows us to navigate the program AST as a graph without being exposed to the details of its relational representation. For instance, `e.getParent+() = f` expresses the property that expression `e` is nested within expression `f` (using QL's "+" syntax for transitive closure).

If all client analyses use `Expr` instead of directly accessing the EDB, we can easily change our data representation later on. For instance, it may not be desirable to record the parent expression directly in the `exprs` table, since toplevel expressions do not have a parent expression. Instead, the parent-child relation could be stored in a separate three-column table `expr_nesting(child,parent,idx)`. The first two columns are foreign keys, so they must refer to properly defined `@expr` values, but there is no requirement that every `@expr` value appears in the first column (or, for that matter, the second column), so expressions without parents can now be modelled.

If we want to switch to this representation, we can simply update the definitions of `getParent` and `getChildExpr` without affecting any client analyses:

```
class Expr extends @expr {
  Expr getParent() { expr_nesting(this, result, _) }
  Expr getChildExpr(int i) { expr_nesting(result, this, i) } ...
```

4.3 Inheritance

Class `Expr` abstracts away from the details of the relational encoding of the AST and is useful for implementing generic syntax tree traversal, but if we want a richer semantic interface we have to implement subclasses of `Expr`. For instance, we could implement a class `EqExpr` to exclusively represent equality checks (and no other expressions):

```
class EqExpr extends Expr {
  EqExpr() { exprs(this, 2, _, _) }
  Expr getLeftOperand() { result = this.getChildExpr(0) }
  Expr getRightOperand() { result = this.getChildExpr(1) }
  string toString() { result = "===" } }
```

The characteristic predicate filters out those expressions that do not have kind 2 (which, in our example encoding, represents equality). We provide getter predicates for the two operands of the equality, further abstracting away from the details of our AST representation, and we override the `toString` predicate to provide a more specialised string representation. Similar classes can be implemented for variable references, literals, and other expressions.

QL classes thus allow us to impose an abstract data type representation on relational data. Since classes can freely overlap, we can even implement multiple representations for the same data. For instance, we could overlay a control flow graph structure on top of the AST by defining a class `CFGNode` that also extends `@expr`, but presents it under a different interface, offering, say, a method `getASuccessor()` to compute call graph successors.

4.4 Overriding

As a practical example of overriding, consider implementing `Expr.isPure`, the member predicate used in Listing 1. Its default implementation in class `Expr` is `none()`, which is a built-in predicate that always fails. In other words, we conservatively assume that all expressions are impure, and override it in subclasses:

```
class Expr extends @expr { predicate isPure() { none() } ...
```

In class `Literal`, we override `isPure` as `any()`, a built-in predicate that always succeeds:

```
class Literal extends Expr { predicate isPure() { any() } ...
```

As another example, equality checks are pure if all of their children are:

```
class EqExpr extends Expr {
  predicate isPure() { forall (Expr c | c = this.getChildExpr(_) | c.isPure()) } ...
```

4.5 Interface vs Implementation

Abstract classes support decoupling interface and implementation even further: while `Expr` implements an interface in terms of one particular set of EDB relations, abstract classes specify only an interface, which may be implemented in multiple different ways.

As an example, assume we want to implement an analysis for JavaScript to find comparisons between expressions with incompatible (dynamic) types, which will always evaluate to `false` at runtime. Assume further that we have implemented a binary predicate `incompatTypes(e, f)` that infers possible types of `e` and `f` and checks whether they are compatible. Using class `EqExpr` defined above, we could implement our analysis as follows:

```
from EqExpr eq, Expr l, Expr r
where l = eq.getLeftOperand() and r = eq.getRightOperand() and incompatTypes(l, r)
select eq, "Operands have incompatible types."
```

Other JavaScript language constructs that compare values in the same way include, e.g., the `switch` statement. If we want to consider them in our query, we could add another disjunct to the `where` part, but this would make it less readable, and we would need to keep extending it for any other equality tests we want to support. Instead, we introduce an abstract class capturing the common interface for all equality tests:

```
abstract class EqualityTest extends ASTNode {
  abstract Expr getALeftOperand(); abstract Expr getARightOperand(); }
```

Like all classes, `EqualityTest` needs a superclass: we choose `ASTNode`, which is a common superclass of `Expr` and `Stmt` defined in the JavaScript QL libraries. The interface defined by `EqualityTest` consists of member predicates to access the left and right operands of the comparison. We allow a single equality test to have multiple left or right operands; e.g., in a `switch`, every `case` is viewed as a right operand of the comparison.

We can implement this interface on `EqExpr` and `SwitchStmt` by introducing new classes that have the same extent as `EqExpr` and `SwitchStmt`, respectively, but extend `EqualityTest`:

```
class EqExprEqualityTest extends EqExpr, EqualityTest {
  Expr getALeftOperand() { result = this.getLeftOperand() }
  Expr getARightOperand() { result = this.getRightOperand() } }

class SwitchEqualityTest extends SwitchStmt, EqualityTest {
  Expr getALeftOperand() { result = this.getExpr() }
  Expr getARightOperand() { result = this.getACase().getExpr() } }
```

The extent of `EqualityTest` now contains all equality expressions and all switch statements, under a convenient interface for our query:

```
from EqualityTest eq, Expr l, Expr r
where l = eq.getALeftOperand() and r = eq.getARightOperand() and incompatTypes(l, r)
select eq, "Operands have incompatible types."
```

To add support for other kinds of equality tests, all we need to do is to define new subclasses of `EqualityTest`; the query need no longer be changed.

4.6 Optimisation

In practice, the translation shown in Figure 2 can produce very inefficient Datalog, particularly when translating virtual calls: the disjunction over all candidates can be quite large, and in many cases the context restricts the receiver variable in such a way that some disjuncts end up always being false, which would lead to a lot of wasted computation if evaluated naively. For example, in the translation shown in Section 3.5, the `query` predicate restricts `e` to `Even`, so the dispatch disjunct $\texttt{Digit.kind}^{\text{disp}}(\texttt{e},\texttt{k})$ can never apply. Another source of inefficiency are superfluous type guards for variables that are already restricted sufficiently by their uses. For instance, the conjunct `string(result)` in `Even.kind` is implied by `result="even"` and hence unnecessary.

One could devise a more sophisticated compilation scheme that avoids this, but we choose to instead perform these optimisations at the Datalog level, keeping the translation as simple as possible: infeasible dispatch disjuncts are simply a special case of formulas that logically contradict other formulas in their context and hence are equivalent to false, while unnecessary type tests are logically implied by other formulas in their context and hence equivalent to true. Both cases can be detected by a form of type inference [15, 33]. We also apply various standard optimisations such as inlining, join ordering and the magic sets transformation [7]; the latter two rely on (compile-time) estimation of (run-time) relation sizes [34].

Ultimately, the example from Section 3.5 is simplified by our optimiser to

```
Even(this) ← range(this, 0, 9) ∧ mod(this, 2, 0).
query(e, k) ← Even(e) ∧ k = "even".
```

5 Case Study

To demonstrate the benefits of QL in implementing static checks, we reimplemented Error Prone (`http://errorprone.info`) in QL. Error Prone is a bug finding tool for Java that integrates with the compiler and checks for common mistakes, reports them and suggests possible fixes. As of version 2.0.4, there are 101 checks. We reimplemented the checks (but not the fix suggestions), ensuring that they pass all the unit tests of the original. This required about one man-month of effort by an experienced QL programmer.

Error Prone is originally implemented in Java, comprising about 10,500 lines of code, not including supporting libraries such as the `javac` Compiler Tree API.[7] Our reimplementation, by contrast, is slightly less than 2,000 lines of code, not including the QL standard library for Java. However, our implementation only covers the checks themselves, not the suggested fixes. Manual inspection suggests that the latter account for about 1,100 lines of code in the Java implementation, leaving 9,400 lines of analysis code.

[7] That is, counting only files in the `src/main/java/com/google/errorprone/bugpatterns` directory.

Listing 2 QL code for detecting nested null checks in Java.

```
// a "==" test where one operand is a null literal
class NullCheck extends EQExpr { NullCheck() { getAnOperand() instanceof NullLiteral } }

// "inner" is nested inside the then−branch of "outer", and both check nullness of "v"
predicate nestedNullCheck(IfStmt outer, IfStmt inner, Variable v) {
  inner.getParent+() = outer.getThen() and
  outer.getCondition().(NullCheck).getAnOperand() = v.getAnAccess() and
  inner.getCondition().(NullCheck).getAnOperand() = v.getAnAccess() }
```

Java is famously verbose, which explains part of the size difference, although QL is overall syntactically quite similar to Java. If we exclude Java `package` declarations and `@Override` annotations (which have no QL counterparts) and `import` statements (the number of which is largely determined by the organisation of the supporting libraries), we can subtract a further 2,800 lines from Error Prone and 100 lines from our implementation. In other words, the Java implementation is 3.5x the size of the QL implementation.

This is mostly because AST traversal and filtering, which require lots of boilerplate code in Java, can be expressed very concisely using recursion and prescriptive typing in QL. For example, Listing 2 shows part of a query for finding incorrect uses of double-checked locking: `NullCheck` picks out comparisons to `null`, and `nestedNullCheck` identifies nested `if` statements that check the same variable for nullness. Recursion is used to check the nesting condition. The casts to `NullCheck` would fail in a descriptive interpretation, since `s.getCondition()` is not a `NullCheck` for most `if` statements `s`. In QL, they act as filters, restricting `outer` and `inner` to those `if` statements that do, in fact, check nullness.

To assess scalability, we ran both the original Error Prone analyses and our reimplementations on a recent snapshot of Apache Hadoop,[8] which is about 1.5 MLoC. Four of the Error Prone checks failed with null pointer exceptions, the remaining 97 finished in 46 seconds. Our reimplementation is about 4x slower, at 201 seconds for the same 97 analyses.[9]

The performance difference is not so much due to the use of Java instead of QL, but to a fundamentally different execution model: while Error Prone checks are performed on a per-file basis, our queries run on a database representing the entire program, and hence are implicitly global. This reflects different usage scenarios: Error Prone is normally run during compilation or as an IDE service, hence instantaneous feedback is important. Our analysis usually runs offline, often even on a dedicated machine, hence a runtime of several minutes on a large code base (and memory requirement of a few gigabytes) is entirely acceptable. Ultimately, these differences are orthogonal to the choice of language, and there is no fundamental obstacle to running local QL analyses on small databases representing one file each.

In summary, this case study shows that QL allows us to quickly implement static analysis checks as concise and scalable queries, though the whole-program approach of our implementation imposes a performance overhead.

As of March 2016, Semmle's static analysis platform offers about 2500 individual analyses for eight languages (C/C++/Objective-C, C#, Cobol, Java, JavaScript, PL/SQL, Python, Scala), all implemented in QL. While some analyses are shallow local checks similar to Error Prone, many crucially depend on whole-program analysis. This is especially true for the dynamically typed languages, where global invariants that a statically typed language would

[8] Commit SHA 855d529 from `https://git-wip-us.apache.org/repos/asf/hadoop.git`.
[9] As measured on an Intel Core i7-4900MQ laptop, with 1GB of heap allocated to the analysis. Timings do not include compilation time for Error Prone and database construction time for QL.

express in the type system instead have to be derived by flow analysis. For instance, our Python analysis suite includes checks for common mistakes such as using undefined attributes or hashing unhashable objects, which sit on top of a whole-program points-to analysis also implemented in QL. As another example, our Java and C++ suites include security analyses based on inter-procedural taint tracking that identifies vulnerabilities to common attacks such as cross-site scripting or SQL injection. All these analyses are in daily use by our customers on multi-million line code bases. A thorough discussion of technical details is out of scope for this paper, but the advantages of using Datalog for static analysis are well-known [35], to which QL adds the benefits of object-oriented modularity and reusability.

Besides implementing static checks, we also use QL for structural analysis to determine dependencies between different parts of a code base that are then visualised in an architectural design tool. Finally, we use QL as a meta-query language to write queries over static analysis results, comparing and correlating results across multiple revisions of the same code base in order to track introductions and fixes of defects over the history of a code base [6].

6 Discussion and Related Work

In this section, we will discuss QL's object-oriented features in the light of popular definitions of object orientation in the literature, and then proceed to survey related work.

6.1 Discussion

The deliberately minimalist examples of Section 2 perhaps make the gap between QL classes and methods and their more traditional counterparts appear wider than it is. Recall that in practice QL programs rely on an extensional database, and here the parallels become quite striking: tuples in a table are records with columns as fields; primary keys uniquely identify tuples, and hence play the role of addresses; foreign keys uniquely reference other tuples, thus acting like references to other records. Hence, a database can be viewed as a strongly typed heap containing a collection of objects with fields that may contain primitive values or references to other objects, where objects with the same layout are collected into tables.

A QL class like `Expr` whose extent is (a subset of) the primary key column of a table thus describes a set of records; its member predicates can access record elements, using the primary key for lookup. Subclasses inherit member definitions and can override them, allowing different implementations of the same operation for different objects, the appropriate implementation being chosen at runtime based on the dynamic type of the object.

Of course, QL classes are not restricted to this particular setting, since classes can be sets of arbitrary values, not just primary keys, and EDB tables can be arbitrary relations, for which there is no parallel in other object-oriented languages.

QL fits the folklore definition of object orientation as "data abstraction plus inheritance": Section 4 shows how QL classes achieve the former by abstracting from the concrete layout of database tables; inheritance in QL is entirely conventional, and, as usual, can be used both for implementing an interface and for code reuse.

A more refined characterisation is given by Cook [11] (building on the classic definition by Wegner [39]), whereby object orientation is support for the dynamic creation and use of objects. An object, in turn, is a first-class, dynamically dispatched *behaviour*, where a behaviour is a collection of named operations, and dynamic dispatch means that different objects can implement the same operations in different ways.

QL satisfies the second half of this definition: classes associate named operations with arbitrary values, and subtyping and virtual dispatch allow different implementations of the

same operations. The first half of Cook's definition is *not* satisfied, as QL programs cannot create new values, which is arguably a natural limitation for a query language. QL also does not support mutable state, and has no notion of object identity as opposed to value identity, but Cook, in fact, argues that these features are not essential to object orientation.

Ullman [37] claims that query languages cannot be "seriously logical and seriously object-oriented at the same time". His argument is framed in the context of a radical reinterpretation of the relational model, where tuples and relations are understood as objects with relational operators as methods. In Ullman's analysis, this would mean that each tuple computed during evaluation is distinct from any previously computed tuple, conflicting with the set semantics of Datalog and its least fixpoint semantics. Furthermore, it is unclear what types to assign to relational operators such union or join in such a setting. Our approach differs significantly from this model: QL classes range over values, not tuples, sidestepping his first point. Relational operators are primitive language constructs, not methods, and hence not typed. In Ullman's view, this might disbar QL from being "seriously" object-oriented, but we have argued that it nevertheless provides the usual benefits of object orientation.

Proposals for integrating object identity and logic programming range from explicitly exposing object identifiers [42] to sophisticated extensions of the underlying logic [24]. Our experience with QL suggests that object identity is not a prerequisite for object orientation.

In summary, QL supports data abstraction and inheritance with dynamic dispatch, widely considered as hallmarks of object orientation. Less conventionally, QL supports overlapping classes and relational member predicates, which are quite natural for a logic query language, while object creation and mutation do not fit this paradigm well and hence are not supported.

6.2 Related Work

Encoding hierarchical data in relational form is conceptually quite straightforward, but querying the encoded data in a traditional relational language is cumbersome. This has been termed the *object-relational impedance mismatch*, and led to calls for replacing the relational model with models directly supporting structured data [5]. We will not discuss the literature on this topic, since QL is based on a completely conventional relational data model. Our approach agrees in spirit, if not in detail, with the so-called Third Manifesto [13], which argues that object orientation should be built on, rather than supplant, the relational model.

Object-oriented extensions of Datalog have been studied in the literature before. Abiteboul et al. [2] propose a language where individual rules for predicates may be associated with classes. They consider three variants of overriding, one of which, termed *static inheritance*, is somewhat similar to QL's approach, although they define overriding at the level of individual rules, not entire predicates. Since their language has no static types they have no concept of rootdefs, instead considering all methods with the right name and arity as dispatch candidates; as we have argued, this makes dispatch highly non-local and brittle in the presence of overlapping classes. They also do not consider multiple inheritance. While for the most part they assume that classes are defined directly by the EDB, they also briefly consider "virtual" classes (first proposed in [1]), which, like concrete QL classes, are defined by a characteristic predicate (though it is unclear whether they can be recursive). Their proposal never seems to have been implemented, making it hard to gauge its practicality.

Extensions of Prolog with subtyping and inheritance have also been proposed [3, 36]. These approaches focus on types for structured terms and on performing unification modulo subtyping between term constructors, which are not available in Datalog.

The idea of storing source code in a database and exploring and analysing it using relational queries goes back at least to Linton [27], who used the INGRES database system.

To overcome the limited expressiveness of standard database query languages, several authors have suggested the use of Prolog [23, 14], which, however, tends to suffer from scalability problems on large code bases. Aiming for a middle ground, Paul et al. [31] propose writing queries in a relational algebra with transitive closure, while Consens et al. [10] employ a subset of Datalog just powerful enough to express properties of paths in graphs.

Many new languages specifically designed for code exploration have been proposed; we discuss just a few examples: ASTLOG [12] focuses on AST traversal, permitting a very efficient implementation. PQL [22] allows more general queries over graphs, while JQuery [21] is based on a full-featured, Prolog-like logic language. JTL [9] only supports querying Java source code, allowing for a very concise and specialised syntax; its expressive power is that of first order logic with transitive closure. Martin et al. [28] propose a DSL for matching sequences of events on objects both statically and dynamically, based on the bddbddb Datalog system [40], which has also been used directly for program analysis.

Our own work in the area started with CodeQuest [20], which compiled Datalog queries to SQL. An early version of QL, which likewise compiled to SQL and only supported analysing Java, was described in [17, 16], where it was presented informally through examples. The present paper provides a more rigorous description of QL including new features such as abstract classes and recursive characters, presents an experimental case study, and (perhaps most importantly) gives a formal semantics. Apart from its potential use in exploring the metatheory of QL, the semantics has already proved useful in clarifying subtle semantic issues such as the type of `this` in characters, which was left implicit in previous work.

The benefits of Datalog for specifying and implementing highly scalable flow analyses [8, 35] have recently been demonstrated on the LogicBlox platform [4], which extends Datalog with support for database updates and creating fresh values. It would be interesting to investigate whether these concepts could be fruitfully combined with QL's object-oriented features.

Graph databases have also been suggested as a basis for source code exploration and analysis. Ebert et al. [18] use the custom graph query language GreQL, while Urma et al. [38] show that the Cypher language of Neo4j (`http://neo4j.com`) allows simple queries to be expressed concisely and evaluated efficiently even over large code bases. Neither language appears to provide abstraction mechanisms like user-defined predicates, making them less well-suited for implementing more complex analyses. Compiling QL to GreQL or Cypher is not possible in general, since they do not seem to provide native support for recursion (besides transitive closure), which is heavily used by typical QL analyses. The four example queries shown by Urma can be written in QL just as concisely (5, 4, 8 and 6 lines, respectively), and execute efficiently (0.4s, 5s, 0.1s, 4s) on a recent snapshot of OpenJDK (2.5MLoC), somewhat faster than the runtimes they report on a similar-sized code base.

Another interesting alternative are metaprogramming languages like Rascal [25], which provide seamless integration of program analyses with code generation and transformation tools. However, their suitability for deep analyses has not been shown yet.

Virtual dispatch in QL is a form of predicate dispatch [19], with class characters as guards. While exhaustiveness is ensured (that is, each call has at least one dispatch candidate), we make no attempt to prevent ambiguity: calls with more than one actual target are fully supported. Also, our translation from QL to Datalog is non-modular and requires reasoning about the entire program, unlike more recent work on predicate dispatch [30].

Prescriptive type systems have been studied in the context of Prolog [41, 26]. As pointed out by Meyer [29], prescriptive type annotations are essentially runtime type checks; hence, in spite of its static type declarations QL is, in some sense, dynamically typed.

7 Conclusion

We have presented QL, an object-oriented dialect of Datalog with classes, subtyping and dynamic dispatch, which we have described both informally and through a translation to plain Datalog: classes are unary predicates representing sets of values; subtyping is set inclusion; and dynamic dispatch resolves calls in the smallest class containing the receiver value. As a typical application of QL, we have shown its use in implementing static checks, and presented a case study highlighting its advantages in this domain over Java. Finally, we have discussed QL's merits as an object-oriented language: while it is missing object creation and mutable state, QL does offer the twin features of abstraction and dynamic dispatch, usually considered to be at the heart of object-oriented programming, without relying on objects in the traditional sense. Apart from QL's practical usefulness, its model of object orientation is an interesting contribution in itself, which, we hope, will spur further discussion of and investigation into the nature of object-oriented programming.

References

1 Serge Abiteboul and Anthony J. Bonner. Objects and views. In *SIGMOD*, 1991.

2 Serge Abiteboul, Georg Lausen, Heinz Uphoff, and Emmanuel Waller. Methods and rules. In *SIGMOD*, 1993.

3 Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *JLP*, 3(3), 1986.

4 Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the LogicBlox system. In *SIGMOD*, 2015.

5 Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In *DOOD*, 1989.

6 Pavel Avgustinov, Arthur I. Baars, Anders S. Henriksen, R. Greg Lavender, Galen Menzel, Oege de Moor, Max Schäfer, and Julian Tibble. Tracking static analysis violations over time to capture developer characteristics. In *ICSE*, 2015.

7 François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, 1986.

8 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, 2009.

9 Tal Cohen, Joseph Gil, and Itay Maman. JTL: The Java tools language. In *OOPSLA*, 2006.

10 Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *ICSE*, 1992.

11 William R. Cook. On understanding data abstraction, revisited. In *OOPSLA*, 2009.

12 Roger Crew. ASTLOG: A language for examining abstract syntax trees. In *DSL*, 1997.

13 Hugh Darwen and C. J. Date. The third manifesto. *SIGMOD Records*, 24(1), 1995.

14 Stephen Dawson, C. R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems. In *PLDI*, 1996.

15 Oege de Moor, Damien Sereni, Pavel Avgustinov, and Mathieu Verbaere. Type inference for Datalog and its application to query optimisation. In *PODS*, 2008.

16 Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. .QL: Object-oriented queries made easy. In *GTTSE*, 2007.

17 Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. .QL for source code analysis. In *SCAM*, 2007.

18 Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. GUPRO—generic understanding of programs. *ENTCS*, 72(2), 2002.

19 Michael D. Ernst, Craig S. Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP*, 1998.

20 Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. *CodeQuest*: Scalable source code queries with Datalog. In *ECOOP*, 2006.

21 Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *AOSD*, 2003.

22 Stan Jarzabek. Design of flexible static program analyzers with PQL. *TSE*, 24(3), 1998.

23 Shahram Javey, Kin'ichi Mitsui, Hiroaki Nakamura, Tsuyoshi Ohira, Kazu Yasuda, Kazushi Kuse, Tsutomu Kamimura, and Richard Helm. Architecture of the XL C++ browser. In *CASCON*, 1992.

24 Michael Kifer and James Wu. A logic for object-oriented logic programming. In *PODS*, 1989.

25 Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *SCAM*, 2009.

26 T. L. Lakshman and Uday S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In *ISLP*, 1991.

27 Mark Linton. Implementing relational views of programs. In *SDE*, 1984.

28 Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: A program query language. In *OOPSLA*, 2005.

29 Gregor Meyer. On types and type consistency in logic programming. Technical Report Informatik Berichte 199, FernUniversität Hagen, 1996.

30 Todd D. Millstein, Christopher Frost, Jason Ryder, and Alessandro Warth. Expressive and modular predicate dispatch for java. *TOPLAS*, 31(2), 2009.

31 Santanu Paul and Atul Prakash. A query algebra for program databases. *TSE*, 22(3), 1996.

32 Teodor C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming*. 1988.

33 Max Schäfer and Oege de Moor. Type inference for Datalog with complex type hierarchies. In *POPL*, 2010.

34 Damien Sereni, Pavel Avgustinov, and Oege de Moor. Adding magic to an optimising Datalog compiler. In *SIGMOD*, 2008.

35 Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In *Datalog Reloaded*, 2010.

36 Gert Smolka and Hassan Aït-Kaci. Inheritance hierarchies: Semantics and unification. *JSC*, 7(3/4), 1989.

37 Jeffrey D. Ullman. A comparison between deductive and object-oriented database systems. In *DOOD*, 1991.

38 Raoul-Gabriel Urma and Alan Mycroft. Source-code queries with graph databases—with application to programming language usage and evolution. *SCP*, 97, 2015.

39 Peter Wegner. Dimensions of object-based language design. In *OOPSLA*, 1987.

40 John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In *APLAS*, 2005.

41 Jiyang Xu and David S. Warren. Semantics of types in logic programming. Technical Report DPS-102, ECRC, Munich, 1990.

42 Carlo Zaniolo. Object identity and inheritance in deductive databases—an evolutionary approach. In *DOOD*, 1989.

Fine-grained Language Composition: A Case Study*

Edd Barrett[1], Carl Friedrich Bolz[2], Lukas Diekmann[3], and Laurence Tratt[4]

1 **Software Development Team, Department of Informatics, King's College London.** `http://soft-dev.org/` `http://eddbarrett.co.uk/`
2 **Software Development Team, Department of Informatics, King's College London.** `http://soft-dev.org/` `http://cfbolz.de/`
3 **Software Development Team, Department of Informatics, King's College London.** `http://soft-dev.org/` `http://lukasdiekmann.com/`
4 **Software Development Team, Department of Informatics, King's College London.** `http://soft-dev.org/` `http://tratt.net/laurie/`

Abstract

Although run-time language composition is common, it normally takes the form of a crude Foreign Function Interface (FFI). While useful, such compositions tend to be coarse-grained and slow. In this paper we introduce a novel fine-grained syntactic composition of PHP and Python which allows users to embed each language inside the other, including referencing variables across languages. This composition raises novel design and implementation challenges. We show that good solutions can be found to the design challenges; and that the resulting implementation imposes an acceptable performance overhead of, at most, 2.6x.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases JIT, tracing, language composition

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.3

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.2.1.1`

1 Introduction

Language composition allows programmers to create systems written in a mix of programming languages. Most commonly, a Foreign Function Interface (FFI) to C is provided so that programs can interact with external libraries. However, other instances of language composition are rare, as crossing the barrier between arbitrary languages is difficult. In many cases, the only way to do so is by having different languages run their parts of the system in separate processes that communicate using (slow) inter-process communication mechanisms. The most common alternative is to use a single Virtual Machine (VM) (e.g. a Java VM), and translate all languages into that VM's bytecode format. This enables finer-grained compositions, but their performance is still generally underwhelming [3].

We believe that there are two different types of friction which make good language compositions difficult: *semantic friction* occurs when an aspect of one language has no

* This research was funded by the EPSRC COOLER (EP/K01790X/1) and LECTURE (EP/L02344X/1) grants.

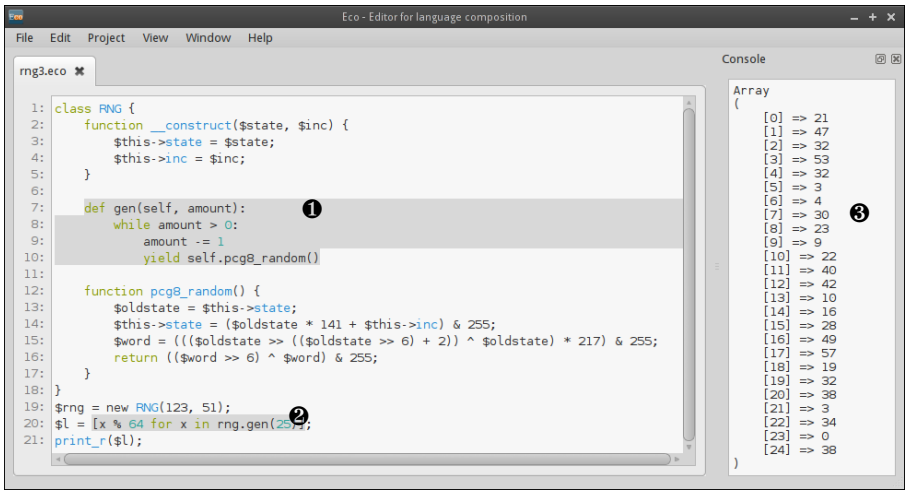


Figure 1 A PCG8 pseudo-random number generator [25] PyHyp program, written in the Eco language composition editor. In this case, the composed PHP and Python program will be exported to PyHyp compatible source code. The outer (white background) parts of the file are written in PHP, the inner (grey background) parts of the file in Python. ❶ A Python language box is used to add a generator method written in Python to the PHP class `RNG`. ❷ A Python language box is used to embed a Python expression inside PHP, including a cross-language variable reference for `rng` (defined in line 19 in PHP and referenced in line 20 in Python). In this case, a Python list comprehension builds a list of random numbers. When the list is passed to PHP, it is 'adapted' as a PHP array. ❸ Running the program pretty-prints the adapted Python list as a PHP array.

equivalent in the other; and *performance friction* occurs when the implementation of one language's behaviour forces the other to execute slowly.

Our hypothesis is that it is possible to reduce the currently accepted levels of friction in language compositions. We believe that the only way to test this hypothesis is through a concrete case study, since friction manifests in different ways in each language composition. We therefore composed together two real-world, widely used languages, Python and PHP, to make a new language composition called PyHyp. At a low-level, PyHyp defines a (somewhat traditional) FFI between PHP and Python that allows cross-language calls and the exchange of data. Building on the FFI, PyHyp then provides the basis for a novel syntactic composition. As shown in Figure 1, a single file can contain multiple fragments of PHP and Python code, and variables can be referenced across different language fragments (e.g. Python code can 'see' PHP variables and vice versa). Unlike approaches which translate one language into another, PyHyp does not alter existing language semantics, nor does it limit users to a subset of either language.

Depending on how one chooses to classify programming languages, Python and PHP can appear similar—most obviously, both are dynamically typed. From our perspective, however, there are a number of tricky differences: PHP has multiple global namespaces which can span multiple files, whereas Python uses one global namespace per file (semantic friction); most of PHP's core data-structures are immutable, whereas many of Python's are mutable (semantic and performance friction); and PHP's sole collection data type is a mapping, whereas Python separates the notion of mappings from that of sequences (semantic and performance friction). As this may suggest, this combination of languages presents a number of design and implementation challenges which have no obvious precedent. We show that PyHyp's solutions to these challenges allow interesting case studies to be implemented.

The practicality of our work rests on two recent developments. First, and most important, is the concept of interpreter composition. The basic idea is to make use of systems which can generate Just-In-Time (JIT) compiled VMs solely from the description of an interpreter (e.g. RPython [6] or Truffle [32]). There are three existing compositions in this style: Prolog and Python [3]; C and Ruby [16]; and C, Ruby, and JavaScript [17]. In essence, each of these systems implements a traditional FFI between its constituent interpreters. All of the systems have good peak performance, but implement simple FFIs. PyHyp defines a much finer-grained FFI between its two languages and also enables syntactic composition between the two languages. The second concept we make use of is language boxes as found in the Eco editor [11], which allow users to naturally write fragments of different languages alongside each other. Note that PyHyp neither extends, nor requires, Eco; however, Eco does hide several tedious details from users.

Although universal answers to our hypothesis are impossible, PyHyp shows that it is possible to create compositions which validate our hypothesis. To summarise, we show that:

1. PyHyp's FFI addresses a number of challenging semantic friction points.
2. Syntactic composition is possible and that practical designs can be found for novel cross-language features.
3. PyHyp's fine-grained language composition has, in the worst case, a performance overhead of 2.6x over its mono-language constituents.

A VirtualBox VM containing repeatable experiments, data and case studies is available at `http://dx.doi.org/10.4230/DARTS.2.1.1`.

2 Background

We assume a basic knowledge of Python syntax and semantics, but not of meta-tracing, interpreter composition, or PHP. In this section we provide overviews of the latter three.

2.1 Meta-tracing

Tracing JIT compilers record hot loops ('traces') in an interpreted program, optimise those traces, and then compile them into machine code [2, 13]. An individual trace is thus a record of one particular path through a program's control flow graph. Subsequent executions which follow that same path can use the machine code generated from the trace instead of the (slow) interpreter. To ensure that the path followed really is the same, 'guards' are left in the machine code at every possible point of divergence. If a guard fails, execution then reverts back to the interpreter.

Meta-tracing JITs have the same basic model, but replace the manually written tracer and machine code generator with equivalents automatically generated from the interpreter itself [24, 29, 33, 4, 6]. The key to good meta-tracing performance is heavily optimised traces. Language implementers can annotate the interpreter provide 'hints' to the meta-tracer to improve the quality of compiled traces. For example, hints can be used to mark parts of the interpreter constant, allowing the trace optimiser to apply constant folding. Similarly, hints can mark a function in the interpreter as 'elidable': given the same inputs, it always returns the same outputs.[1] The optimiser can then replace calls to (slow) elidable functions with

[1] Unlike pure functions, elidable functions can have idempotent side-effects (e.g. caching). The user is responsible for guaranteeing that the relationship between inputs and outputs is maintained.

(fast) checks on input values, substituting the output values in place of the function call. Identifying parts of an interpreter amenable to such hints requires the author's knowledge of both the semantics of the language being implemented and common idioms of use.

In this paper we use RPython, the main extant meta-tracing language. RPython is a statically typed subset of Python with a type system similar to Java's. Unlike seemingly similar languages (e.g. Slang [18] or PreScheme [22]), RPython is more than just a thin layer over C: it is, for example, fully garbage collected and has several high-level data types (e.g. lists and dictionaries). Despite this, VMs written in RPython have performance levels which far exceed traditional interpreters [6]. The specific details of RPython are generally unimportant in most of this paper, and we do not dwell on them: we believe that one could substitute any reasonable meta-tracing language (or its cousin approach, self-optimising interpreters with dynamic partial evaluation [32]) and achieve similar results.

2.2 Interpreter Composition

Interpreter composition involves 'glueing' together two or more existing interpreters such that each can utilise the other. Assuming that both interpreters are written in the same language, a basic composition is simple: interpreter A needs to import interpreter B and then call appropriate functionality in B. A more sophisticated composition will define matters such as data type conversion, and how and when execution passes between its constituent interpreters. Achieving the desired composition can require adding entirely new glue code, and/or invasively modifying the constituent interpreters.

Since, traditionally, interpreters are slow, composing them tends to worsen performance [3]. Fortunately, composed interpreters are as amenable to meta-tracing – and its close cousin dynamic partial evaluation [32] – as their constituent interpreters. Existing examples of such compositions include Python and Prolog [3] and Ruby, C and JavaScript [17]. Both systems have good peak performance.

In the rest of this paper, we use 'interpreter' to mean the source-code of the system that is used to produce an executable binary 'VM'. In practice, most readers can consider the terms 'interpreter' and 'VM' to be interchangeable with only a small loss of precision.

2.3 PHP

PHP is a language used widely for server-side webpage creation. Originally intended as a language to glue together CGI programs written in C, it gradually evolved into a complete programming language. Largely due to this gradual evolution, PHP features a number of design decisions that appear unusual when viewed from the perspective of other imperative languages such as Python or Ruby. As one example that appears later in the paper shows, many of PHP's primitive data types, including arrays, are immutable. We give further details on such features as needed.

PHP's syntax is Perl-esque, taking influence from the Unix shell (e.g. variables start with a '$') and C (e.g. basic control structures). The following (contrived) example shows most of the syntax needed to understand this paper's examples:

```
1 $i=1;                 // Assign 1 to variable $i
2 $j=&$i;               // Create a reference to var $i
3 $a=array(3, 4, 5);    // Create a list-like array
4 $b=array("bob"=>45);  // Create a dictionary-like array
5 foreach ($a as $c) {  // Iterate over the array
6   echo $c . " ";      // Print out (in order) 3 4 5
7 }
8 $o=new C();           // Create a new object
9 $o->m($i);            // Call method m
```

```
10 $s = <<<EOD           // Start multiline string
11   A string
12   with multiple lines
13 EOD;                  // End multiline string
```

3 PyHyp

PyHyp is a language composition of PHP and Python, implemented by composing two existing RPython interpreters: HippyVM (for PHP) and PyPy (for Python). PyPy is an industrial-strength Python interpreter which can be used as a drop-in replacement for CPython 2.7.8; HippyVM is a partially complete PHP 5.4 interpreter. PyHyp is a 'semantics preserving' composition in the sense that it adds behaviour to both Python and PHP, but does not alter or remove existing behaviour.

PyHyp programs currently start by executing PHP code. There is no deep reason for this choice, and one could easily allow it to start by executing Python code instead. Since they start as normal PHP programs, 'raw' PyHyp programs require `<?php ... ?>` tags around the entire file. In the interests of brevity, we omit these in all code listings.

We stage our explanation of PyHyp as follows: the design and implementation of its FFI (Section 4); its support for syntactic composition (Section 5); and finally cross-language variable scoping (Section 6).

4 PyHyp FFI

PyHyp defines an FFI which is the core upon which advanced functionality is later built. A simple example of PHP code using the FFI to interact with Python is as follows:

```
1 $random = import_py_mod("random");
2 $num = $random->randrange(10, 20);
3 echo $num . "\n";
```

The code first imports Python's `random` module into PHP (line 1) before calling the Python `randrange(x, y)` function to obtain a random integer between 10 and 20 (line 2) which is then printed (line 3). The only explicit use of the FFI in this example is the call to `import_py_mod`. However, the FFI is implicitly used elsewhere: the PHP integers passed as arguments to `randrange` are converted to Python integers, and the result of the function is converted from a Python integer to a PHP integer. As this may suggest, the FFI is two-way and Python code can also call PHP.

4.1 FFI Design

Many parts of PyHyp's FFI are fairly traditional, while some are unusual due to the semantic friction between Python and PHP. To the best of our knowledge there has not previously been an FFI between Python and PHP, so our solutions are necessarily novel.

4.1.1 Data Type Conversions

All FFIs have to define data type conversions between their constituents. Since primitive data types in both PHP and Python are immutable, PyHyp directly maps them from one language to the other (e.g. a PHP integer is transformed into a Python integer). Arbitrary user objects cannot be directly mapped and are instead wrapped in an *adapter* which allows the other interpreter to work transparently with the underlying foreign instance. A PHP object, for example, appears to Python code as a normal Python object, whose attributes

and methods can be accessed, introspected etc. Passing an adapter back to the language from which it was created simply removes the adapter. Adapters are immutable, only ever pointing to single object in their lifetime; the trace optimiser is then extremely effective at removing the overhead of repeated adaptations.

Collection data types are more involved. Python separates the notion of a list (i.e. resizeable array) from that of a dictionary (i.e. hashmap). In contrast, PHP has a single dictionary type called, somewhat confusingly, an array. PHP arrays are therefore also used wherever 'lists' are required. This presents an interesting case of semantic friction. Python lists and dictionaries can both be sensibly adapted in PHP as arrays. PHP arrays passed to Python, in contrast, are ambiguous: should they be adapted as lists or dictionaries? It is easy to design schemes which can be dangerously subverted. For example, a PHP array which 'looks like' a list might seem best adapted as a Python list, but later mutation to its keys (e.g. adding a string key) can turn it into something which is clearly not a list.

The only consistent design is therefore to default to adapting PHP arrays as Python dictionaries. However, users often know that a given PHP array is, and always will be, equivalent to a list. Therefore, PHP arrays adapted as Python dictionaries have an additional method `as_list`, which re-adapts the underlying array as a Python list. Whenever operations on the list adapter are called, a check is made to see whether the underlying PHP array is still list-like; if it is not (e.g. because a non-integer key has been added) an exception is raised.

In general, converting an adapted object to its 'host' language simply requires removing the adapter. The one exception is a Python list which has been passed to PHP, adapted as a PHP array, and which is subsequently returned back to Python. Since our data-conversion rules dictate that PHP arrays are adapted as Python dictionaries, Python code expecting a PHP array would get a surprising result if the PHP array returned was unwrapped directly to a Python list (rather than a dictionary). Thus a Python list adapted as a PHP array and then returned to Python has a special Python dictionary adapter. Only if `as_list` is called on that adapter is the underlying Python list returned.

4.1.2 Mutability

PHP data types are immutable except for objects (which are mutable in the same way as objects in Python) and references. Immutable data types often use copy-on-write semantics. For example, appending to an array creates a copy with an additional element at the end. Operations on references – mutable cells which typically point to immutable data – are passed onto the underlying datum, which may be replaced. For example, appending to a reference which points to an array mutates the reference to point to the newly copied array.

Since it is common to wrap PHP arrays in a PHP reference, and since Python's expectations are that such data types are mutable, PyHyp does not directly adapt arrays: instead, arrays are replaced by references to arrays, which are then adapted. Put another way, PHP arrays always appear to Python as mutable collections, whether adapted as lists or dictionaries, and those mutations are visible to PHP code as well.

4.1.3 Cross-language Calls

Both Python and PHP functions can be adapted and passed to the other language where they can be called naturally. There are, however, two cases of semantic friction: Python functions with keyword arguments; and PHP's pass-by-reference mechanism.

Simplifying slightly, Python functions accept zero or more mandatory, ordered arguments, and zero or more unordered, keyword arguments, each of which has a default value. Function calls must pass parameters for each ordered argument and then zero or more keyword arguments. The following example shows such a function and an example call:

```
1 def fmturi(host, path, scheme="http", frag="", query=""):
2   uri = "%s://%s%s" % (scheme, host, path)
3   if query: uri += "?%s" % query
4   if frag: uri += "#%s" % frag
5   return uri
6 fmturi("google.com", "/", frag="q=ecoop")
```

While PHP allows parameters to have default values, arguments must be passed in order, and there is no notion of keyword arguments. To enable PHP to call Python functions such as `fmturi`, PyHyp adds a global PHP function `call_py_func(f, a, k)` where `f` is an adapted Python function, `a` is an array of regular arguments, and `k` is an array of keyword arguments. From PHP, one can thus emulate the function call from line 6 as follows:

```
1 call_py_func($fmturi, array("google.com", "/"), array("frag" => "q=ecoop")).
```

By default, PHP parameters are pass-by-value but function signatures can mark their arguments as being pass-by-reference by prepending the parameter name with `&`. When a function with such a parameter is called, a reference is created which points to the argument passed (if it is not already a reference). Thus one can write a PHP function which swaps the contents of the variables passed to it:

```
1 function php_swap(&$x, &$y) {
2   $tmp = $y;
3   $y = $x;
4   $x = $tmp;
5 }
6 $a = 10; $b = 20;
7 php_swap($a, $b);
8 echo "$a $b\n"; // prints "20 10"
```

As this example shows, the code calling `php_swap` has no control over whether it is passing arguments with pass-by-value or pass-by-reference semantics—indeed, calling `php_swap` updates `$a` and `$b` so that they point to the newly created references. Since PyHyp needs to allow Python functions to be used as drop-in replacements for PHP functions, we need a notion of pass-by-reference for Python function arguments. This is tricky since Python has no explicit notion of a reference.

PyHyp takes a two-stage approach to reducing this case of semantic friction. First, we introduce an explicit `PHPRef` adapter into Python which represents a mutable PHP reference. `PHPRef`s support two explicit operations: `deref()` returns the value inside the reference; and `store(x)` mutates the reference to point to `x`. Second, we add a Python decorator `php_decor` which takes a keyword argument `refs` which specifies (as a sequence of argument indices) which arguments are pass-by-reference. With this, we can write a Python swap function as follows:

```
1 @php_decor(refs=(0, 1))
2 def py_swap(a, b):
3   tmp = a.deref()
4   a.store(b.deref())
5   b.store(tmp)
```

Although it may be tempting to think that `PHPRef`s should be transparent in Python, as they are in PHP, we found such a scheme to be impractical: it would require changing every possible part of the Python language and implementation where one can read or write to variables. Explaining the effects to users would be extremely challenging, as would changing

the implementation. In PyPy, for example, we estimate that this would involve changing around 100 separate locations.

Because `PHPRef`s are explicit, calling a PHP function with pass-by-reference arguments from Python is possible but, inevitably, somewhat clunky. Pass-by-reference arguments must be explicitly passed a `PHPRef` object; other object types lead to a run-time exception. Thus Python can call `php_swap` as follows:

```
xref, yref = PHPRef(x), PHPRef(y)
php_swap(xref, yref)
x, y = xref.deref(), yref.deref()
```

4.2 PyHyp FFI Internals

Until now, we have detailed the language PyHyp presents to the user. We now consider PyHyp's internal implementation details. PyHyp required modifying both HippyVM and PyPy. We added modules to both HippyVM (`pypy_bridge`) and PyPy (`hippy_bridge`), which encapsulate most of PyHyp's behaviour. Most of the common behaviour resides in the `pypy_bridge` module, though it could just as easily reside in `hippy_bridge`. Some behaviour is implemented by invasively modifying existing HippyVM or PyPy code.

4.2.1 Data Type Conversion

Viewed from a suitable level of abstraction, both HippyVM and PyPy implement their respective languages using broadly similar data type hierarchies: a root data type class – not entirely coincidentally, called `W_Root` in both interpreters – from which all objects inherit. Generally PyHyp adapters extend, directly or via a subclass, one of the `W_Root`s.

We added methods to both root data type classes: a `to_py` method to every PHP data type; and a `to_php` method to every Python data type. Calling `to_py` on a PHP datum creates a Python adapter (and vice versa for `to_php`). The default implementations of `to_py` and `to_php` return generic adapters, but other data types override them to return specialised adapters. Calling `to_py` / `to_php` on an adapter simply returns the adapted datum. The only exception is calling `to_py` on a Python list which has been adapted as a PHP array; rather than returning the Python list itself, PyHyp is forced to return a special (new) variant of a Python dictionary (see Section 4.1.1).

The generic adapters – one each in HippyVM and PyPy– simply forward attribute lookups, method calls, and the like onto the adapted object. PyHyp then defines a number of specialised adapters: 10 additional Python adapters and 8 additional PHP adapters. Some of the special adapters expose different behaviour to the user (e.g. collection data types), whereas some deal with low-level differences between data types in the VM (e.g. PyPy's layout requires separate adapters to be defined for functions and methods). As PyPy uses storage strategies to optimise collection data types [5], adapted PHP collections create Python collection instances that use PyHyp-specific strategies rather than subclassing `W_Root`.

The code for adapters is self-contained and relatively simple. Together, the PHP and Python adapters are just under 1400LoC, of which 400LoC implements new storage strategies.

4.2.2 Mutability

In order to make PHP arrays mutable from Python, PyHyp requires PHP arrays passed to Python to be wrapped in a reference. However, simply adding a reference when an adapter

is created would lead to mutations from Python not being seen by PHP. PyHyp therefore handles arrays specially: the following two examples give a flavour of this.

The `ARG_BY_PTR` PHP opcode organises function arguments prior to a function call, and is where arrays passed to a call-by-reference function argument have their storage in the PHP frame replaced by a reference. PyHyp adds a special case to this opcode: every PHP array passed to a Python function is treated as if it was being passed to a call-by-reference function argument. This ensures that the PHP frame observes mutations from Python.

Similarly, when Python loads an array from an adapted PHP object's attribute, PyHyp must replace the attribute with a reference. This ensures that the parent object observes mutations from Python. Implementing this is fairly simple, as we reuse an existing function in HippyVM which can lookup an attribute and turn non-references into references (used to implement PHP's standard `x=&$y->z` behaviour). We add a new flag to this function, since we only want to turn arrays (and not other data types) into references.

4.2.3 Cross-language Calls

For the most part, cross-language calls are simple to implement. PyHyp is careful to ensure that the necessary glue code is optimised and does not obstruct meta-tracing's natural cross-language inlining. Most importantly, this requires annotating the relevant RPython functions as being unrollable, so that they dynamically specialise themselves to the number of parameters passed by the user.

The `php_decor` decorator is implemented as a normal Python (not RPython) class. When the decorator is applied with the `refs` keyword argument, the argument indices are stored in a normal, user visible attribute of the function object. When the function is adapted for PHP, the adapter loads the indices, which are later used by the PHP interpreter to determine which parameters are to be passed by reference.

4.2.4 Transparency

Ensuring that adapters are as transparent as possible inevitably requires invasive modifications of HippyVM and PyPy. We were helped by the fact that both interpreters centralise all the behaviour we wished to modify. For example, implementing identity transparency in PyPy is easy, as identity checks are not handled directly but handed over to an object's `is_w` method which compares the current object with another for identity equality. PyHyp adapters override this method so that if the objects being compared are both adapters of the same type, the identity check is forwarded on to the underlying adapted objects. The `is_w` method of `W_PHPGenericAdapter` (the class representing an adapted PHP object in PyPy) shows this idiom:

```
1 def is_w(self, other):
2   if isinstance(other, W_PHPGenericAdapter):
3     return self.w_php_obj is other.w_php_obj
4   return False
```

5 Syntactic Composition

Traditionally, FFIs have made the implicit assumption that the source code of each language involved is kept in different files. In this section, we show how PyHyp allows PHP and Python code to be used within a single file. To avoid tedious duplication of explanation, we concentrate our explanation on embedding Python into PHP, though PyHyp also allows PHP to be embedded into Python.

5.1 Functions

At a low-level, PyHyp provides simple support for embedding Python inside PHP (and vice versa) as strings. For example, the `compile_py_func` function takes a string containing a single Python function and returns a Python function object that is adapted as a callable PHP instance. The following example embeds a Python function inside PHP and calls it to produce a random number between 0 and 10:

```
$src = <<<EOD
def randnum(n):
  import random
  return random.randrange(n)
EOD;
$randnum = compile_py_func($src);
echo $randnum(10) . "\n";
```

In this paper we mostly use `compile_py_func`, though `compile_py_func_global` can be used to compile a Python function and put it in PHP's global function namespace.

Although the `compile_*` functions are called at run-time, they are surprisingly fast. First, due to PHP and Python being simple languages to compile, both HippyVM and PyPy have efficient compiler implementations. Second, when JIT compiled, PyHyp caches the bytecode output of `compile_*` calls. Re-evaluating a function thus produces a new (cheap) function object while reusing the (expensive) bytecode object which underlies it.

5.2 Methods

PyHyp supports inserting Python methods into PHP classes via the `compile_py_meth(c, f)` function, where the Python method source `f` is compiled and inserted into the class named by the string `c`. However, this feature must be used carefully because HippyVM's implementation takes advantage of the fact that PHP's classes can be statically compiled. This means that, by the time a program has started running, normal PHP classes cannot be altered. This poses a problem for PyHyp because its syntactic embedding is currently performed at run-time. We work around this by enclosing a PHP class inside curly braces, which delays its compilation until run-time. We also require that all calls to `compile_py_meth(c, ...)` immediately follow the definition of `c` and that the class and all such calls be surrounded by curly brackets. This also affects sub-classes of `c`, whose execution must also be delayed by curly brackets (though not necessarily at the same point in the source code).

PHP supports a Java-like public/private/protected method access scheme, but Python has no concept of private or protected methods.[2] To model this, PyHyp extends the `php_decor` decorator (see Section 4.1.3) with an optional extra argument `access` which accepts a string access value (`"public"`, `"private"`, or `"protected"`). Similarly, the option `static` boolean argument allows static methods to be denoted. These arguments can be combined to specify that a Python method is e.g. protected and static.

5.3 Using Eco to Express Embeddings

For simple uses, the `compile` functions are tolerable as-is, but they tend to obfuscate embedded code, especially in multi-level embeddings (e.g. PHP inside Python inside PHP), where string escaping becomes onerous.

[2] Python attributes prefixed by two underscores have their names mangled, but are otherwise publicly accessible.

```
# Create PHP grammar referencing Python (and vice versa)
python = Grammar("python.eco")
php = Grammar("PHP+Python", "php.eco")
python.add_alternative("atom", php)
php.add_alternative("top_statement", python)
php.add_alternative("class_statement", python)
php.add_alternative("expr", python)
# Create Python expressions-only grammar
python_expr = Grammar("Python expressions", "python.eco")
python_expr.change_start("simple_stmt")
php.add_alternative("expr", python_expr)
```

Listing 1 Composing PHP and Python grammars in Eco. `Grammar(n, p)` loads a grammar named `n` from path `p`. `change_start` changes the start rule of a grammar. `g1.add_alternative(r, g2)` adds a new alternative to the rule `r` in grammar `g1` to the start rule of grammar `g2`.

In order to make PyHyp's syntactic composition more palatable, we make use of the Eco editor [11]. Eco allows users to compose grammars and to write composed programs. In essence, one takes a context-free grammar X and adds a reference from a rule $R \in X$ to another grammar Y. When entering language X into Eco, the user can switch to entering language Y by creating a language box. Language boxes delineate when one language ends and another starts, but do not introduce ambiguity or complexity into a parser. Eco parses as the user type, and thus always has access to parse trees for all languages involved in a composition. However, from the user's perspective, editing in Eco feels like a normal text editor, except when creating a language box or moving between nested language boxes (a fairly rare occurrence).

We first had to enable users to write PyHyp programs. Since Eco comes with a Python grammar, we only had to add a PHP grammar (which we adapted, with minor modifications, from Zend PHP) and a PHP lexer. We then used Eco's Python interface to express the join points between the two grammars (see Listing 1). Simplifying slightly, in PHP one can add Python language boxes wherever PHP statements or expressions are valid; and in Python one can add PHP language boxes wherever a Python expression is valid. From the users perspective, this means that when using Eco for PyHyp, they create a new `PHP+Python` file and start typing PHP. When they want to insert a Python function they insert a `Python+PHP` language box; when they want to insert a Python expression they insert a `Python expression` language box.

We then had to implement an exporter from parse trees relative to the composed `PHP+Python` composed grammar to PyHyp-compatible input. The exporter automatically inserts `compile_*` functions, follows `compile_py_meth`'s restrictions, and escapes arbitrarily deeply nested language boxes. For example, the `gen` Python language box in the `RNG` class in Figure 1 is exported as follows:

```
{
class RNG { ... }
compile_py_meth("RNG", "def gen(self, amount):\n \\
                        while amount > 0:\n \\
                          amount -= 1\n \\
                          yield self.pcg8_random()");
}
```

Figure 1's second language box (line 20) is more interesting. PyHyp has no explicit interface for embedding Python expressions. Instead, the expression is encoded as a callable PHP instance which is compiled and immediately called:

```
$l = call_py_func(compile_py_func("f = lambda: [x % 64 for x in rng.gen(25)];"));
```

Figure 2 Cross-language exception handling in PyHyp, showing that the stacktrace presents entries from within language boxes correctly. As an example, the first line of the stacktrace should be read as follows: "The 0th entry in the stacktrace relates to the exceptions.eco file, line 2, within the `h` function".

In the rest of the paper we use the term 'language box' to refer to an embedded language fragment irrespective of whether Eco is used or not.

5.3.1 Exceptions

When a native exception crosses the language boundary, PyHyp adapts it, before re-raising the exception. For example, if PHP calls Python code which raises the Python exception `ZeroDivisionError`, then the exception will appear to PHP as a generic `PyException`. As with all other adapters, an adapted exception crossing back to its native language (e.g. a `PyException` which percolates back to Python) simply has its adapter removed.

However, adapters on their own do not solve a crucial problem: cross-language stacktraces. By default, both HippyVM and PyPy can only print out their own frames in stacktraces, which makes cross-language exceptions seem to appear out of thin air. Equally frustratingly, frames which represent inner language boxes report incorrect line numbers, as each language box's frame assumes it starts at line 1.

PyHyp fixes both problems. First, we altered HippyVM and PyPy's stacktrace functions to call each other for their appropriate frames. Second, we added two arguments to PyHyp's compilation functions (e.g. `compile_py_func`) to allow Eco to pass the file and line offset the compiled text is relative to. The stacktrace routines then use this information to adjust the reported locations. The end result is that cross-language stacktraces are just as informative as mono-language stacktraces, as can be seen in Figure 2.

6 Cross-language Scoping

For syntactic composition to be useful, we believe that users must be able to reference variables across language boxes. PyHyp therefore allows both Python and PHP to reference variables in the other language, making syntactic composition significantly more powerful and usable. This raises a novel design challenge: what are sensible cross-language scoping rules? The major challenge is to deal with both language's expectations surrounding global

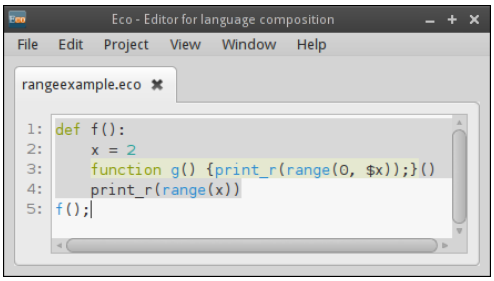


Figure 3 Cross-language scoping with nested language boxes: each language expects to see its own global scope. Lexical scoping suggests that `x` referenced on lines 3 and 4 should bind to the definition on line 2. Since PHP and Python's `range` functions are incompatible, the PHP language box on line 3 must reference a different `range` function to that on line 4. However, both the PHP and Python language boxes wish to use PHP's `print_r` function on lines 3 and 4. PyHyp's scoping rules respect these desires and this example prints out 0, 1, 2, 0, 1.

namespaces. We eventually settled upon scoping rules that are relatively easily explained, and which impose a mostly lexical system. This is no small matter, as PHP and Python have significantly different scoping rules.

In this section, we first define a simplified version of the scoping rules in each language (since PHP and Python share neither common semantics nor terminology, we do our best to homogenise the description). We then define PyHyp's additional rules for PHP referencing Python and vice versa, before explaining how these rules are implemented.

6.1 PHP and Python's Namespace Semantics

PHP has separate global namespaces for functions, classes, constants, and variables. Any given name can appear in multiple namespaces, as syntactic context uniquely identifies which namespace is being used (e.g. `new x()` references a class, whereas `x()` references a function). PHP has no concept of modules, textually 'including' files in similar fashion to C headers. Thus the global namespaces span all PHP files.[3] The namespaces for functions, classes, and constants can have new names added to them dynamically, but existing names cannot be removed or changed. In contrast, names can be added or removed to the namespace for variables at will. Each function then defines a local scope; variable lookups within a function first search the local scope before searching the global variable namespace.

Python modules each have a single 'global' namespace, to which names can be added and removed. Functions define their own local namespace. Lexical lookups (for names defined in the current or parent function's scope) are determined statically (the set of local names cannot be modified); global lookups are performed dynamically.

Neither PHP nor Python has Scheme-esque closures[4]: nested functions can read a parent function's variables but writes are not shared between the two. In the interests of brevity, we consider PHP and Python's variable scoping rules to be local and global, but not lexical.

[3] Although PHP 5.3 introduced a mechanism for defining non-global namespaces, this does not effect our explanation.
[4] For our purposes, we consider what PHP calls a 'closure' to be a first-class anonymous function.

6.2 PyHyp's Cross-language Scoping Rules

The use of cross-language scoping in Figure 1 may suggest that cross-language scoping design is a matter of applying traditional language design principles. Figure 3 shows a more challenging example, where modern expectations about lexical scoping and PHP and Python's global namespaces appear to clash.

PyHyp resolves this issue with the following simple design. First, PHP and Python's local scoping rules remain unchanged. Second, we split the search for variables which are not bound in the current language box into two distinct phases: the 'recursive' phase, and the 'global' phase. The recursive phase searches language boxes (from inner to outer) for a matching variable definition. Failing this, the global phase searches the global namespace(s) of the current language box's language; if no match is found, it then searches the global namespace(s) of the other language.

The recursive phases for PHP and Python are conceptually identical. Python's global phase is simple, but PHP's is complicated by its multiple global namespaces. If the search originated from PHP, then only the appropriate namespace for the syntactic context is used (e.g. if, syntactically, a function was looked up, only the function namespace is searched); if the search originated in Python the namespaces are searched in the following order: global functions, classes, then constants.

Performance reasons led us to make one small adjustment to the PHP global phase search. PHP has the ability to lazily load classes; every failed class lookup triggers a (fairly slow) check for user-defined lazy loading mechanisms. In mono-PHP this is a sensible mechanism, as in practice either a class is found in the namespace, or it is lazily loaded, or an error is raised. However, since cross-language scoping frequently checks for the existence of names that do not, and will never, exist, the cumulative performance effect can be frustrating. Since disabling lazy loading would break many existing PHP applications, we tweaked PyHyp's scoping rules. The search for a Python name in PHP's global namespaces is 'sticky': if a name x was found to be (say) a class on the first search in a given scope, it will only ever return a class on subsequent searches (i.e. if a function x is later added to the functions namespaces, it will not be returned as a match in that scope). This small loss of dynamicity increases performance in some benchmarks by around 50%.

Using Figure 3 as a concrete example, we can see how these rules apply in practice. First, consider the PHP variable reference `$x` on line 3. There is no binding of `x` in the PHP language box so when the code is executed a recursive phase search commences: the parent (Python) language box is inspected and a binding found on line 2. The `range` function reference on line 3 starts with the same pattern: a recursive phase search looks in the Python language box for a binding but fails. A PHP global phase search then commences; since `range` was syntactically referenced as a function, only the global function namespace is searched, where a match is then found. The `print_r` function reference on line 3 follows the same pattern. The reference to a `range` function in the Python language box at line 4 starts with a recursive phase search which looks into the parent PHP language box's for a suitable binding (i.e. a name starting with a '`$`') but fails. It then does a global phase search in Python's global namespace, finding Python's built-in `range` function. The `print_r` reference on line 4 is more interesting. A recursive phase search fails to find a match. A global phase search then searches in Python's global namespace and fails before trying PHP's global namespaces, starting with functions, where a match is found.

PyHyp's scoping rules also work well in corner-cases (e.g. PyHyp deals with PHP's superglobals sensibly). Note that the scoping rules of both languages are partly or wholly dynamic: that is, in some situations, bindings can be changed at run-time. PyHyp's scoping

rules maintain PHP's and Python's dynamic lookup properties since some programming idioms (particularly in PHP, but also in Python) rely on adding or removing bindings.

6.2.1 Implementation

To add PyHyp's scoping rules to HippyVM and PyPy, we first needed to connect language box scopes together at run-time, and then intercept the locations where PHP and Python global variables are read and written to.

Connecting language box scopes is made relatively simple by the fact that each is constructed by a `compile_*` function (see Section 5.1). The outer part of any PyHyp program is, by definition, a PHP language box and every other language box is nested inside that. Thus any call to `compile_py_*` implicitly receives a reference to the PHP frame from which it was called. The reference is then stored in a `PHPScope` object, which PyHyp attaches to the Python function object being created; nested Python functions inherit a `PHPScope` object from their parent function, so that multiply nested functions can still access outer language boxes. Similarly, when a PHP language box is nested inside Python, a `PyScope` object is created and placed inside a PHP function's bytecode object. This simple scheme means that, from any PHP or Python frame, one can walk a chain from the current to the outermost language box.

To actually search outer language box's scopes, we have to modify those parts of HippyVM and PyPy which perform global lookups. In HippyVM, we modify the 3 separate functions on the main interpreter object which perform searches of functions, classes, and constants, as well as the `lookup_deref` function on frames which lookups up variables. An elided version of the `locate_function` function – which searches for a PHP function `n` – shows the small scale of such modifications:

```
1  def locate_function(n):
2    py_scope = self.topframeref().bytecode.py_scope
3    if py_scope is not None:
4      ph_v = py_scope.ph_lookup_local_recurse(name)
5      if ph_v is not None: return ph_v
6      ph_v = self.lookup_function(name)
7      if ph_v is not None: return ph_v
8      ph_v = py_scope.ph_lookup_global(name)
9      if ph_v is not None: return ph_v
10   else:
11     func = self.lookup_function(name)
12     if func is not None: return func
13   self.fatal("Call to undefined function %s()" % name)
```

In essence, the original function consisted of lines 11 and 12; PyHyp adds lines 1–9. When a PHP function performs a global function lookup, and that PHP function is nested inside a Python language box (lines 1 and 2) then a local phase search is performed (lines 4 and 5). If unsuccessful, the global phase search then commences: first the PHP global function namespace is searched (lines 6 and 7) before Python's global namespace is searched (lines 8 and 9). The modifications made to PyPy are identical in idiom, modifying the two opcodes (`LOAD_GLOBAL` and `STORE_GLOBAL`) which read and write global variables.

Since PyHyp's rules are highly dynamic, we rely heavily on use of Self-style maps (see [9]) to turn most name lookups from (slow) dictionary lookups of names into (fast) constant lookups. HippyVM originally made only limited use of maps in its global namespaces: we altered it to use maps extensively.[5] We also used maps for the sticky namespace search. Looking up a variable in a global phase search in a PHP scope returns an integer representing

[5] This optimisation also helps plain HippyVM, as it significantly improves the performance of programs such as MediaWiki and phpBB that use the `$GLOBALS` superglobal.

unknown (i.e. this name has not previously been searched for in this context), class, function, constant, or not found (i.e. a search was previously done and no match was found). After tracing, virtually all global lookups turn into a small number of (quick) attribute guards, avoiding (slow) dictionary lookups.

7 Experiment

To understand PyHyp's performance characteristics, we define three classes of benchmarks: small microbenchmarks, large microbenchmarks, and permutation benchmarks. Benchmarks come in four variants: mono-language PHP (henceforth 'mono-PHP'); mono-language Python ('mono-Python'); composed PHP and Python where PHP is the 'outer' language ('composed-PHP') and where Python is the 'outer' language ('composed-Python'). We run these benchmarks on several PHP and Python implementations.

7.1 Benchmarks

Our small microbenchmarks, focus on single language features in isolation, and are useful for identifying low-level pinch points. Each of our small microbenchmarks consist of two parts. In most, an outer loop repeatedly calls an inner function (e.g. the *total_list* benchmark's inner function takes a list of integers and sums them). In the remainder, an outer function generates elements, and an inner function consumes them. In the composed variants, the inner and outer components are implemented in different languages.

Some small microbenchmarks cannot be implemented in all variants. The *ref_swap* benchmark measures the performance of operations on PHP references and `PHPRefs` (see Section 4.1.3) and thus has no mono-Python variant. Benchmarks which require putting PHP methods into Python classes are currently not supported by PyHyp. The complete list of small microbenchmarks can be found in Appendix A.

Our large microbenchmarks aim to measure performance more broadly. We use four 'classic' benchmarks: *Fannkuch* counts permutations by continually flipping elements in a list [1]; *Mandel* plots an ASCII representation of the Mandelbrot set into a string buffer[6]; *Richards* models an operating system task dispatcher[7]; and *DeltaBlue* is a constraint solver [27]. To create composed variants of these benchmarks, we took the mono-language variants and replaced each function with an implementation in the other language. The composed-PHP variants of Richards and DeltaBlue are thus PHP 'shell' classes containing many Python methods (33 and 75 respectively), with variables referenced between languages and data repeatedly crossing from PHP to Python (Figure 4 shows the mono and composed-PHP variants of Richards alongside each other). In other words, Richards and DeltaBlue are designed to heavily test PyHyp's cross-language performance. In contrast, Fannkuch is a single function, and so the composed variant consists of a single Python function embedded in PHP. This serves as a rough baseline, since we would expect that the composed variant has roughly the same performance as PyPy. Mandel also started off life as a single function, but we made modifications to make a more interesting benchmark: we split the innermost loop into a separate function; made the function's parameters pass-by-reference; and made the function modify these references during execution. Since Mandel uses references, there

[6] `Zend/bench.php` in the Zend distribution of PHP.
[7] `http://www.cl.cam.ac.uk/~mr10/Bench.html`

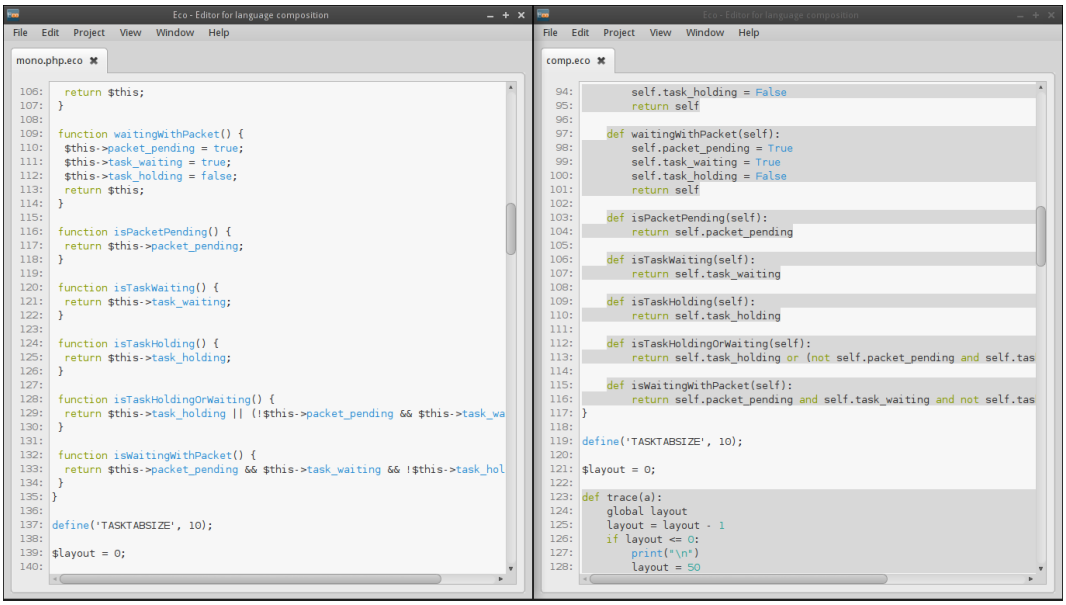


Figure 4 The mono-PHP and composed-PHP variants of Richards side by side. The composed-PHP variant of the benchmark contains a 'shell' PHP program with PHP classes whose methods are Python language boxes. Global variables remain defined in PHP, so that the benchmarks also include an element of cross-language scoping.

is no mono-Python variant. There is no composed-Python variant of either Richards or DeltaBlue, since PHP methods cannot yet appear inside Python classes.

The permutation benchmarks are designed to uncover whether some parts of a program are faster in one or other language. Using the mono-PHP DeltaBlue benchmark as a base, we created 79 permutations, each with one PHP function replaced by a Python equivalent. We then compare the timings of each permutation to the original mono-PHP benchmark. For brevity, we henceforth refer to permutation number x as p_x.

7.2 Methodology

Each benchmark was run on the following VMs (in alphabetical order): CPython, the standard interpreter for Python; HHVM, a JIT compiling VM for PHP; HippyVM; PyHyp$_{\text{PHP}}$, which is PyHyp running composed-PHP variants; PyHyp$_{\text{Py}}$, which is PyHyp running composed-Python variants; PyHyp$_{\text{mono}}$, which is PyHyp running mono-PHP variants; PyPy; and Zend, the standard interpreter for PHP. The versions used for each of these VMs is shown in Table 1. Note that PyHyp$_{\text{PHP}}$, PyHyp$_{\text{Py}}$ and PyHyp$_{\text{mono}}$ are all the save VM, and we use the terms to be clear about which benchmarks PyHyp is running.

For each benchmark and VM pair, we ran 5 fresh processes, with each process running 50 iterations of the benchmark. We then used the bootstrapping technique described in [21] to derive means and 99% confidence intervals for each pairing. Since we could not always determine when a given VM had warmed up, we made no attempt to remove any iterations from the process: thus our timings include warmup. All timings are wall-clock with a sub-microsecond resolution.

All experiments were run on an otherwise idle 4GHz Core i7-4790 CPU and 32GiB RAM machine, running Debian 8. We disabled turbo mode and hyper-threading in the BIOS. We

Table 1 The VM versions used in this paper.

Interpreter	Version(s)
CPython	2.7.10
HHVM	3.4.0
HippyVM	git #2ae35b80
PyPy	2.6.0
Zend	5.5.13
PyHyp	Based on above HippyVM/PyPy versions.

used a tickless Linux kernel, disabled Intel p-states, and ensured that the CPU governor was set to maximum performance mode. All VMs were built with the system GCC (4.9.2). We did not interfere with the garbage collection of any of the VMs, which run as normal.

7.3 Results

Table 2 shows the results of our microbenchmarks relative to the composed PyHyp variant. Absolute timings are shown in Table 4 in the Appendix. Starting with the simplest observations, we can see that Zend and CPython (C-based interpreters) are slower than HHVM, HippyVM, and PyPy (JIT-based VMs). Small to medium benchmarks tend to flatter meta-tracing, and so HippyVM and PyPy outperform HHVM. PyPy is nearly always faster than HippyVM, reflecting the greater level of engineering PyPy has received.

$\text{PyHyp}_{\text{mono}}$'s results are very similar to HippyVM's, though a few cases run slightly slower on PyHyp. For walk_list and DeltaBlue, some missing optimisations in PyHyp's scoping lookups cause undue bloat in the optimised traces. instchain and sum_meth_attr in contrast have identical traces except for a small portion of their headers: this seemingly small difference has a surprisingly large run-time effect which we do not fully understand.

PyHyp is generally faster than HippyVM on the composed-PHP benchmarks. This is largely due to moving code from PHP (slower HippyVM) to Python (faster PyPy) and the ability that meta-tracing has to naturally inline code across both languages. PyHyp is in most cases slower than PyPy, as we would expect, because of the additional overhead of adapters and cross-language scoping. Although meta-tracing naturally optimises the vast majority of these operations away, a few inevitably remain, and their cumulative effect is often noticeable, even though it is not severe. On the geometric mean $\text{PyHyp}_{\text{PHP}}$ is only around 20% slower on average than PyPy, and no individual $\text{PyHyp}_{\text{PHP}}$ benchmark is more than 2.2x slower. The composed-Python benchmarks have a similar overall average to the composed-PHP benchmarks, though several benchmarks are slower. By comparing traces from the composed-PHP and composed-Python benchmarks, we were able to identify several missing optimisations in HippyVM that are likely to account for most such slowdowns: redundant comparisons in logical operators; many more allocations in PHP iterators; and more allocations when appending to PHP lists.

In some cases, the timings for composed variants running on PyHyp are virtually identical to the mono-language variants running on PyHyp's constituents (e.g. smallfunc on HippyVM, $\text{PyHyp}_{\text{PHP}}$ and PyPy are all roughly 1x). For small benchmarks, we would expect any well-written RPython VM to compile virtually identical traces, and such bench-

Table 2 Microbenchmark timings relative to $\text{PyHyp}_{\text{PHP}}$. Note that $\text{PyHyp}_{\text{PHP}}$ and PyHyp_{Py} are the same VM, but running composed-PHP and composed-Python benchmark variants respectively.

Benchmark	CPython	HHVM	HippyVM	$\text{PyHyp}_{\text{PHP}}$	PyHyp_{Py}	$\text{PyHyp}_{\text{mono}}$	PyPy	Zend
instchain	33.451 ±0.0679	9.547 ±0.0096	0.912 ±0.0011	1.000		1.116 ±0.0012	0.675 ±0.0007	36.471 ±0.1577
l1a0r	86.017 ±0.0179	4.052 ±0.0020	1.368 ±0.0004	1.000	1.360 ±0.0003	1.359 ±0.0003	1.340 ±0.0106	38.778 ±0.0078
l1a1r	83.803 ±0.1407	2.980 ±0.0038	1.306 ±0.0017	1.000	1.303 ±0.0016	1.303 ±0.0016	1.140 ±0.0022	39.111 ±0.1272
lists	8.047 ±0.0139	0.931 ±0.0036	0.975 ±0.0020	1.000	0.560 ±0.0012	0.978 ±0.0021	0.497 ±0.0010	14.626 ±0.0377
ref_swap		8.393 ±0.0006	1.000 ±0.0002	1.000	0.700 ±0.0001	1.000 ±0.0001		53.320 ±0.0040
return_simple	110.409 ±0.1104	7.049 ±0.0019	1.000 ±0.0001	1.000	0.778 ±0.0001	1.000 ±0.0001	0.889 ±0.0001	84.724 ±0.0645
scopes	133.487 ±0.0493	15.023 ±0.0018	4.511 ±0.0025	1.000	0.929 ±0.0005	4.495 ±0.0013	1.000 ±0.0001	152.608 ±0.0131
smallfunc	187.132 ±0.1488	13.078 ±0.0010	1.000 ±0.0001	1.000	0.750 ±0.0000	1.000 ±0.0001	1.000 ±0.0001	230.818 ±0.0145
sum	317.479 ±0.2718	19.362 ±0.0014	0.999 ±0.0001	1.000	0.750 ±0.0001	1.000 ±0.0003	0.874 ±0.0001	418.485 ±0.0921
sum_meth	341.850 ±1.3274	24.106 ±0.0280	0.999 ±0.0001	1.000		1.000 ±0.0001	0.874 ±0.0002	447.472 ±0.4913
sum_meth_attr	131.469 ±0.7392	17.915 ±0.1052	0.999 ±0.0061	1.000		1.131 ±0.0065	0.904 ±0.0057	145.365 ±0.8321
total_list	19.230 ±0.0145	2.245 ±0.0008	0.864 ±0.0002	1.000	1.508 ±0.0004	0.858 ±0.0005	0.587 ±0.0003	33.667 ±0.0633
walk_list	5.060 ±0.0071	0.406 ±0.0005	0.779 ±0.0011	1.000	1.601 ±0.0026	1.010 ±0.0018	1.080 ±0.0015	10.647 ±0.0677
deltablue	16.528 ±0.0707	671.482 ±2.9041	4.325 ±0.0212	1.000		4.507 ±0.0214	0.457 ±0.0026	144.149 ±2.6843
fannkuch	20.582 ±0.0226	3.342 ±0.0025	1.848 ±0.0007	1.000	1.891 ±0.0005	1.878 ±0.0005	1.005 ±0.0004	14.387 ±0.0128
mandel		0.791 ±0.0056	0.921 ±0.0005	1.000	0.493 ±0.0001	0.999 ±0.0003		7.241 ±0.0188
richards	26.902 ±0.0189	11.897 ±0.0088	0.853 ±0.0010	1.000		0.887 ±0.0007	0.488 ±0.0005	24.207 ±0.0236
Geometric Mean	52.743 ±0.0341	6.940 ±0.0047	1.222 ±0.0006	1.000	0.963 ±0.0003	1.277 ±0.0006	0.813 ±0.0007	55.549 ±0.0692

marks show this effect. However, in some cases where we expected identical performance for both PyHyp and its constituents, the composed variant is faster: lla0r, l1a1r, and smallfunc. Indeed, the crucial parts of the traces were identical between the two VMs in all these cases. Further exploration showed that RPython's machine code generator occasionally emits less than optimal code (in this case unnecessary x86-64 `MOV`s) that account for the difference. We do not understand the precise reason for this, but it seems plausible it is a limitation of the current register allocator. We have reported our findings to the RPython developers.

Table 3 shows the results from the permutations experiment. The majority of permutations are statistically indistinguishable from mono-PHP; most of the remainder are close enough in performance to be of little interest. Four permutations, however, show substantial differences: p_2, p_5, p_6 and p_7 all perform much better than in mono-PHP. We now describe the reasons for these cases.

p_2 swaps the `OrderedCollection` class's constructor which performs a single action, assigning an array (in PHP) or list (in Python) to the `elms` attribute. The seemingly innocuous change of moving from a PHP array to a Python list has a big impact on performance simply because PyPy's lists are are far more extensively optimised than HippyVM's (see [5]). This provides indirect evidence of the importance of making adapters immutable (see Section 4.1.1): even though p_2 operates extensively on adapters, their costs after trace optimisation are extremely small.

p_5, p_6, and p_7 are all similar in nature. Ultimately, and perhaps surprisingly, the slowdown is due to Hippy using a tracing garbage collector. Because of PHP's copy-on-write semantics, arrays are conceptually copied on every mutation. Zend (the traditional PHP implementation) is able to optimise away many of these writes by making use of its reference counting garbage collector. When a mutation operation occurs on an array with a reference

Table 3 DeltaBlue permutations in PyHyp, with absolute times (in seconds) and relative timings (to mono-PHP DeltaBlue run on PyHyp). Greyed-out cells indicate that the confidence intervals overlap. Bold entries indicate that there is more than a 25% relative performance difference.

P1: 0.246s ±0.0005 1.000× ±0.0029	P17: 0.246s ±0.0004 1.000× ±0.0028	P33: 0.246s ±0.0005 0.999× ±0.0028	P49: 0.246s ±0.0005 1.002× ±0.0028	P65: 0.246s ±0.0005 1.001× ±0.0029
P2: 0.120s ±0.0003 0.490× ±0.0015	**P18: 0.250s ±0.0005 1.015× ±0.0029**	P34: 0.246s ±0.0004 1.000× ±0.0026	P50: 0.246s ±0.0006 1.001× ±0.0032	**P66: 0.247s ±0.0004 1.006× ±0.0026**
P3: 0.240s ±0.0004 0.978× ±0.0026	**P19: 0.245s ±0.0004 0.999× ±0.0026**	P35: 0.246s ±0.0006 1.000× ±0.0031	P51: 0.246s ±0.0004 1.000× ±0.0027	P67: 0.246s ±0.0005 1.001× ±0.0027
P4: 0.249s ±0.0005 1.015× ±0.0030	**P20: 0.245s ±0.0004 0.998× ±0.0026**	P36: 0.246s ±0.0005 1.002× ±0.0030	P52: 0.246s ±0.0004 1.000× ±0.0026	P68: 0.246s ±0.0005 1.000× ±0.0030
P5: 0.132s ±0.0002 0.538× ±0.0014	P21: 0.246s ±0.0004 1.001× ±0.0027	P37: 0.246s ±0.0005 1.001× ±0.0028	**P53: 0.248s ±0.0004 1.008× ±0.0027**	**P69: 0.251s ±0.0004 1.021× ±0.0028**
P6: 0.131s ±0.0002 0.533× ±0.0013	P22: 0.247s ±0.0005 1.005× ±0.0029	P38: 0.246s ±0.0005 1.000× ±0.0028	P54: 0.246s ±0.0005 0.999× ±0.0028	**P70: 0.248s ±0.0005 1.008× ±0.0028**
P7: 0.175s ±0.0003 0.710× ±0.0020	**P23: 0.244s ±0.0005 0.993× ±0.0027**	P39: 0.246s ±0.0005 0.999× ±0.0027	P55: 0.247s ±0.0005 1.004× ±0.0028	**P71: 0.242s ±0.0004 0.986× ±0.0025**
P8: 0.246s ±0.0006 1.002× ±0.0032	P24: 0.246s ±0.0005 1.000× ±0.0029	P40: 0.246s ±0.0005 1.000× ±0.0030	P56: 0.247s ±0.0006 1.005× ±0.0031	P72: 0.246s ±0.0005 1.001× ±0.0028
P9: 0.246s ±0.0005 1.000× ±0.0029	P25: 0.246s ±0.0005 1.002× ±0.0029	P41: 0.245s ±0.0005 0.995× ±0.0028	P57: 0.245s ±0.0005 0.999× ±0.0028	P73: 0.246s ±0.0005 1.000× ±0.0027
P10: 0.246s ±0.0004 1.000× ±0.0026	P26: 0.246s ±0.0004 1.001× ±0.0027	**P42: 0.248s ±0.0004 1.011× ±0.0027**	**P58: 0.245s ±0.0004 0.999× ±0.0026**	**P74: 0.248s ±0.0004 1.011× ±0.0027**
P11: 0.246s ±0.0005 1.000× ±0.0027	P27: 0.247s ±0.0006 1.006× ±0.0031	P43: 0.247s ±0.0005 1.005× ±0.0029	**P59: 0.248s ±0.0006 1.011× ±0.0032**	**P75: 0.251s ±0.0004 1.021× ±0.0027**
P12: 0.246s ±0.0004 1.001× ±0.0027	P28: 0.247s ±0.0005 1.004× ±0.0029	**P44: 0.248s ±0.0005 1.011× ±0.0031**	P60: 0.247s ±0.0005 1.005× ±0.0030	P76: 0.246s ±0.0004 1.002× ±0.0025
P13: 0.246s ±0.0005 1.000× ±0.0029	P29: 0.246s ±0.0006 1.000× ±0.0032	**P45: 0.248s ±0.0005 1.011× ±0.0027**	**P61: 0.248s ±0.0005 1.009× ±0.0028**	**P77: 0.244s ±0.0005 0.992× ±0.0027**
P14: 0.248s ±0.0005 1.009× ±0.0029	P30: 0.246s ±0.0006 1.000× ±0.0031	**P46: 0.270s ±0.0005 1.100× ±0.0029**	P62: 0.246s ±0.0005 1.000× ±0.0027	P78: 0.246s ±0.0003 0.999× ±0.0025
P15: 0.246s ±0.0004 0.999× ±0.0026	P31: 0.246s ±0.0004 1.000× ±0.0027	**P47: 0.267s ±0.0004 1.085× ±0.0027**	**P63: 0.245s ±0.0004 0.999× ±0.0026**	P79: 0.246s ±0.0004 0.999× ±0.0026
P16: 0.246s ±0.0004 1.000× ±0.0026	P32: 0.246s ±0.0005 1.000× ±0.0028	P48: 0.247s ±0.0005 1.003× ±0.0030	P64: 0.245s ±0.0005 0.998× ±0.0028	

count of 1, the array is mutated in place, as the change cannot be observed elsewhere. HippyVM in contrast does not use reference counting, and does not know exactly how many pointers to an array exist at any given point. Checking every pointer in the run-time system would be prohibitively expensive, so HippyVM approximates Zend's optimisation with a 'unique' flag on array references. Various operations can remove uniqueness, but arrays in unique references can be optimised in the same manner as Zend arrays with a reference count of 1. Taking p_5 as a concrete example, we can see the subtle effects of this optimisation in HippyVM. p_5 swaps the `OrderedCollection` class's `size` method, which simply calls `count` (in PHP) or `len` (in Python) on a PHP array stored in an attribute. Since the `count` function is call-by-value, HippyVM optimises the copy that of the array that should occur by simply dropping its unique flag; later mutations thus must therefore copy the array. However, this is not directly why PyHyp is faster than HippyVM in p_5. When the `OrderedCollection` class's `size` method is moved to Python, PyHyp's mutability semantics (see Section 4.1.2) cause `size` to be a pass-by-reference function, thus meaning that the PHP reference does not lose uniqueness, and less copying is then required.

7.4 Threats to Validity

Benchmarks are only ever a snapshot of certain performance characteristics of a system, and we do not pretend that they necessarily tell us about program performance in a more general setting. Our experiments also make no attempt to account for JIT warmup (for reasons explained in Section 7.2). Removing JIT warmup would thus 'improve' the perceived timings of VMs such as PyHyp which perform JIT compilation. Since it is also known that RPython VMs have relatively poor warmup [6], the likely effect of our decision is to make PyHyp look worse relative to other VMs. We consider this a better trade-off than trying to make other VMs look worse relative to PyHyp.

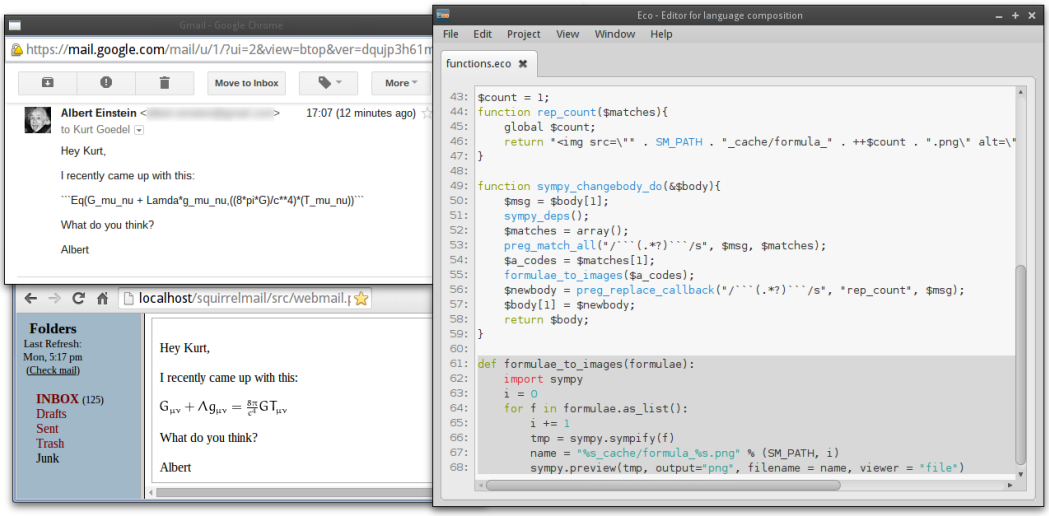


Figure 5 Example mails sent with our extended version of SquirrelMail. We extended this PHP mail client such that it can visualise mathematical formulae using the SymPy Python library. A portion of the plug-in code is shown in the right.

8 Case Studies

8.1 Using CFFI in PHP

PHP does not have a built-in C FFI, whereas Python does via the `cffi` module. PHP code can thus use PyHyp to access `cffi`, acquiring a C FFI by default. For example the following elided example shows PHP using `cffi` to call the Unix `clock_gettime` function:

```
1  $cffi = import_py_mod("cffi");
2  $ffi = new $cffi->FFI();
3  $ffi->cdef("double _clock_gettime_monotonic();");
4  $csrc = <<<EOD
5    double _clock_gettime_monotonic(){
6      struct timespec ts;
7      if ((clock_gettime(CLOCK_MONOTONIC, &ts)) == -1)
8        err(1, "clock_gettime error");
9      return ts.tv_sec + ts.tv_nsec * pow(10, -9);
10   }
11 EOD;
12 $ffi->set_source("_example", $csrc);
13 $C = $ffi->dlopen(null);
14 echo "Monotonic time: " . $C->_clock_gettime_monotonic() . "\n";
```

8.2 A SquirrelMail Plugin

SquirrelMail is a venerable PHP web mail client. We used PyHyp to add a SquirrelMail plug-in that uses the Python SymPy library. This is intended to show that PyHyp can be used to add Python modules to relatively large existing systems. In essence, the plug-in recognises mathematical formulae between triple backticks, and uses SymPy to render them in traditional mathematical notation. Formulae in incoming emails are automatically rendered; users sending emails with such formulae can preview the rendering before sending. Figure 5 shows the plug-in in use, and the core parts of the code within Eco.

The `sympy_changebody_do` function is called by SquirrelMail's `message_body` hook (which is also called upon viewing a message), receiving the content of the email as an argument. A regular expression finds all occurrences of formulae between backticks (line 53)

and passes them to the Python `formulae_to_images` function. This then uses SymPy to convert the formulae to images (numbered by their offset in the array/list) into the directory pointed to by the PHP constant `SM_PATH` (lines 61–68), and uses the URL of the image in-place of the textual formula (line 56).

8.3 System Language Migration

We expect that one of the key uses of syntactic language composition is system language migration, where systems are slowly migrated from language A to B in small stages. Instead of having to rewrite whole modules or sub-systems, syntactic language composition offers the possibility of migrating one function at a time. A full case study is far beyond the scope of this paper, but we implicitly modelled this technique when creating the DeltaBlue and Richards benchmarks, where we translated each PHP method into Python, leaving only 'shell' PHP classes, global functions and variables. As Section 7.3 clearly shows, the resulting performance is at worst 2x of its mono-language variant, which we believe makes system language migration plausible for the first time.

9 Discussion

To give an approximate idea of PyHyp's size, some rough metrics are useful. The `pypy_bridge` module – in which the majority of PyHyp is implemented – adds around 2KLoC. Aside from this we added around 0.25KLoC and 0.2KLoC to the existing HippyVM and PyPy interpreter code respectively. 5KLoC of new unit tests were added. On a fast build machine (4GHz Core i7) PyHyp takes about 45 minutes to build. We estimate that implementing PyHyp took around 7 person months.

A succinct summary of our experiences of creating PyHyp is: implementing what we wanted was fairly easy; making what we implemented run fast was somewhat easy; but working out what we wanted to implement was often hard. The latter point may surprise readers as much as it has surprised us. There are two main reasons for this.

First, there is little precedent for fine-grained syntactic language composition. Most existing language compositions are either extremely crude (per process compositions) or have design decisions implicitly imposed upon them (translating into another VM's bytecode). We therefore faced a number of novel language design issues, and used gradually larger case studies to help us iterate our way to good solutions, sometimes exhausting what felt like every possible alternative. Cross-language scoping is a good example of this: we tried many possibilities before settling on the scheme described in Section 6.

Second, it is difficult, and probably impossible, for any single person to be truly expert in every language and implementation involved in a composition. We sometimes had to base initial designs on (hopefully) intelligent guesses about one or the other languages' semantics or implementations. We then tried as hard as we could to break the resulting design. It rapidly became clear to us that in large languages such as PHP and Python, there are many corner-cases, sometimes little used, which need to be considered. PHP references caused us more headaches than any other language feature. At first we ignored them, and then we failed to appreciate their pervasive nature. It took us considerable effort to understand them well enough to make sense of their place within PyHyp. While we do not pretend to be experts in every aspect of either PHP or Python, we can recommend this route to anyone who wishes to understand the nooks and crannies of a language and its implementation.

Once we had settled upon a good design, we rarely had substantial difficulty in modifying HippyVM or PyPy to implement it. The relatively small size of the additional / changed code

in the composition is a reasonable proxy for this. Similarly, the very nature of meta-tracing meant that most cross-language optimisations came without any extra work on our part. Cross-language variable scoping was the only feature that required substantial optimisation effort on our part, including to HippyVM itself.

9.1 Generalising from the Case Study

PyHyp is, to the best of our knowledge, the first fine-grained language composition. Although we are cautious about over-generalising our results, we believe that some of the lessons embodied in PyHyp may be relevant for future fine-grained language compositions.

Most obviously, despite the rather different run-time properties of PHP and Python, PyHyp's performance is close enough to HippyVM and PyPy to be usable. While we would like to claim credit for all of this, most of the benefit comes from meta-tracing: only in a few places did we have to add PyHyp-specific optimisations. We expect languages even more disparate than PHP and Python to still achieve fairly good performance using meta-tracing.

Our use of adapters meant that most interactions between PHP and Python required little or no effort on our part to compose together satisfactorily. We expect this to generalise to most other compositions. Adapters were also the key to resolving seemingly major semantic data-type incompatibilities between (mostly immutable) PHP and (mostly mutable) Python. The techniques we used are likely to be relevant to compositions involving languages that are more rigorously immutable than PHP.

Finally, despite the archaic nature of PHP and Python's scoping rules, we were able to design good cross-language scoping rules. Most modern languages have embraced lexical scoping, and compositions involving them will require less contortions than PyHyp.

10 Related Work

There has been a long-standing desire for language composition (see e.g. [10]), and many flavours have developed since then. Extensible languages (e.g. [19, 20, 8]) aim to grow a language as required by a user. However, the base language places restrictions on what extensions are possible (e.g. due to parsing restrictions) and performant [31]. Translating one language into another (with e.g. Stratego [7]) removes many of the limitations on what is expressible, but full-scale translations are complex (e.g. [15]) and typically suffer the same performance issues as extensible languages. However, for small use cases, or where performance is not important, either approach can work well.

FFIs are the most common approach to composing languages, but their performance is typically poor due to their inability to inline cross-language calls. The next most common mechanism is to target an existing high performance VM (typically HotSpot). However, since such VMs can only optimise those programs they expect to commonly see. Languages which step even slightly outside this mould perform poorly. For example, Java programs have often excellent performance on HotSpot, but Python programs on HotSpot generally run slower than with simple C-based interpreters [28, 6].

Our aim in this paper has been to show that fine-grained syntactic language composition is possible and performant. We make no claims about the formal properties of the resulting composition as the practical challenges identified in this paper are already substantial. There is already a small body of work on formalising language composition, such as an investigation of the COM architecture [30], and an abstract framework for specifying the operational semantics of multi-language embeddings [23]. There are also partial formal semantics for

languages such as Python and PHP (see e.g. [26, 12]). We welcome future work formalising fine-grained compositions.

As the case studies show, PyHyp is at least somewhat usable, but we are under no illusions that it is an industrial strength product. There are many interesting directions for further exploration, such as experimenting with cross-language inheritance [14].

11 Conclusions

In this paper we introduced PyHyp, a fine-grained syntactic composition of PHP and Python implemented by combining together meta-tracing interpreters. We consider that PyHyp validates our hypothesis that programming languages can be composed in a finer-grained manner than previously thought possible or practical. Not only does PyHyp introduce novel concepts such as cross-language variable scoping, but its performance is close enough to its mono-language cousins to encourage use of such a system. Inevitably, some of PyHyp's details are specific to the particular pair of languages it composes. However, many of the techniques that PyHyp embodies – the use of interpreter composition with meta-tracing; some of the design choices surrounding cross-language scoping – are likely to be of use to future language compositions.

Acknowledgements We thank Armin Rigo for adjusting RPython to cope with some of PyHyp's demands, and advice on Hippy; Ronan Lamy and Maciej Fijałkowski for help with Hippy; Jasper Schulz for help with cross-language exceptions; Alan Mycroft for insightful thoughts on language composition; and Martin Berger, Darya Kurilova, and Sarah Mount for comments.

References

1 Kenneth R. Anderson and Duane Rettig. Performing Lisp analysis of the Fannkuch benchmark. *Lisp Pointers*, 7(4):2–12, Oct 1994.

2 Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI*, pages 1–12, Jun 2000.

3 Edd Barrett, Carl Friedrich Bolz, and Laurence Tratt. Approaches to interpreter composition. *COMLAN*, 44(C), March 2015.

4 Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: A trace-based JIT compiler for CIL. In *OOPSLA*, pages 708–725, Mar 2010.

5 Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Storage strategies for collections in dynamically typed languages. In *OOPSLA*, pages 167–182, Oct 2013.

6 Carl Friedrich Bolz and Laurence Tratt. The impact of meta-tracing on VM design and implementation. *Science of Computer Programming*, 98, Part 3:408–421, Feb 2015.

7 Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *SCICO*, 72(1–2):52 – 70, 2008.

8 Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible grammars for language specialization. In *Database Programming Languages*, pages 11–31, Aug 1993.

9 Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *OOPSLA*, Sep 1989.

10 Thomas E. Cheatham. Motivation for extensible languages. *SIGPLAN*, 4(8):45–49, Aug 1969.

11 Lukas Diekmann and Laurence Tratt. Eco: A language composition editor. In *SLE*, pages 82–101, Sep 2014.

12 Daniele Filaretti and Sergio Maffeis. An executable formal semantics of PHP. In *ECOOP*, pages 567–592, 2014.

13 Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *VEE*, pages 144–153, Jun 2006.

14 Kathryn E. Gray. Safe cross-language inheritance. In *ECOOP*, pages 52–75, jul 2008.

15 Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA*, pages 231–245, Oct 2005.

16 Mathias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. Dynamically composing languages in a modular way: Supporting C extensions for dynamic languages. In *Modularity*, March 2015.

17 Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. High-performance cross-language interoperability in a multi-language runtime. In *DLS*, pages 78–90, 2015.

18 Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *OOPSLA*, pages 318–326, Oct 1997.

19 Edgar T. Irons. Experience with an extensible language. *CACM*, 13(1):31–40, Jan 1970.

20 Gregory F. Johnson and C. N. Fischer. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In *POPL*, pages 141–151, Jan 1985.

21 Tomas Kalibera and Richard Jones. Quantifying performance changes with effect size confidence intervals. Technical Report 4-12, University of Kent, Jun 2012.

22 Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp Symb. Comput.*, 7(4):315–335, Dec 1994.

23 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *TOPLAS*, 31(3):12:1–12:44, Apr 2009.

24 James George Mitchell. *The design and construction of flexible and efficient interactive programming systems*. PhD thesis, Carnegie Mellon University, Jun 1970.

25 Melissa E. O'Neil. PCG: A family of simple fast space-efficient statistically good algorithms for random number generation, 2015.

26 Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: The full Monty. In *OOPSLA*, pages 217–232, 2013.

27 Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *SPE*, 23(5):529–566, 1993.

28 Aibek Sarimbekov, Andrej Podzimek, Lubomir Bulej, Yudi Zheng, Nathan Ricci, and Walter Binder. Characteristics of dynamic JVM languages. In *VMIL*, pages 11–20, Oct 2013.

29 Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic native optimization of interpreters. In *IVME*, pages 50–57, Jun 2003.

30 Kevin J. Sullivan, Mark Marchukov, and John Socha. Analysis of a conflict between aggregation and interface negotiation in Microsoft's component object model. *TOSE*, 25(4):584–599, Jul 1999.

31 Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *TOPLAS*, 30(6):1–40, Oct 2008.

32 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Onward!*, pages 187–204, 2013.

33 Alexander Yermolovich, Christian Wimmer, and Michael Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. In *DLS*, pages 79–88, Oct 2009.

A Small Microbenchmarks

Section 7.1 outlined the small microbenchmarks used in our experiment. This appendix lists and describes each of the small microbenchmarks. The following microbenchmarks consist of an outer loop calling an inner function:

l1a0r The inner function takes an integer which is decremented to zero in a loop. Nothing is returned.
l1a1r The inner function takes an integer which is decremented to zero in a loop. After every decrement, the value is added to a sum total. The sum is returned.
ref_swap The inner function swaps its two arguments using references. Since Python has no support for references, there is no mono-Python variant of this benchmark.
return_simple The inner function returns a constant integer.
scopes The inner function takes a parameter and adds it to a variable from an outer scope. In the composed variants, the scope lookup crosses language boxes.
smallfunc The inner function takes three arguments a, b, and c, and returns $a + b * c$.
sum The inner function takes five arguments, sums them, and returns the result.
sum_meth As *sum*, except the sum is computed and returned by a method. The method belongs to an object which is allocated once and re-used.
sum_meth_attr As *sum_meth*, except that the result is stored to an attribute of the object.
total_list The inner function sums the elements of a list/array passed as an argument.

The following microbenchmarks consist of one function generating elements which another function consumes:

instchain A nested chain of objects is constructed and consumed in a loop. Each object in the chain has an attribute storing an integer. One function constructs the chain, another walks it summing the integers. These functions are called in a loop. In the composed variants, the outer loop is in one language and the construct and walk functions (including utility methods) are in the other.
lists One function constructs a list of integers, and another iterates over the list, summing its elements. These functions are called in a loop to repeatedly construct and sum lists. In the composed variant, the summing function is written in one language, with all other parts written in the other.
list_walk One function creates a linked list while the other function walks the list. Each element in the list is a three element tuple (x, y, n) where x and y are integers and n is a pointer to the next element, or the string `"end"` for the final element. As the list is walked, a counter is incremented by $y - x$. In the composed variant, the list creation and walking functions are in a different language from the outer loop.

The l1a0r, l1a1r, lists, smallfunc, and list_walk microbenchmarks are ports of benchmarks from [3]. All other small microbenchmarks were created specifically to test PyHyp.

Table 4 Absolute microbenchmark timings.

Benchmark	CPython	HHVM	HippyVM	PyHyp$_{\text{PHP}}$	PyHyp$_{\text{Py}}$	PyHyp$_{\text{mono}}$	PyPy	Zend
instchain	11.323 ±0.0208	3.232 ±0.0016	0.309 ±0.0002	0.338 ±0.0003		0.378 ±0.0002	0.228 ±0.0001	12.345 ±0.0519
l1a0r	15.965 ±0.0011	0.752 ±0.0003	0.254 ±0.0000	0.186 ±0.0000	0.252 ±0.0000	0.252 ±0.0000	0.249 ±0.0019	7.198 ±0.0005
l1a1r	16.473 ±0.0162	0.586 ±0.0001	0.257 ±0.0000	0.197 ±0.0002	0.256 ±0.0000	0.256 ±0.0000	0.224 ±0.0003	7.689 ±0.0166
lists	3.871 ±0.0021	0.448 ±0.0015	0.469 ±0.0005	0.481 ±0.0008	0.269 ±0.0003	0.470 ±0.0005	0.239 ±0.0002	7.036 ±0.0136
ref_swap		2.573 ±0.0001	0.306 ±0.0001	0.306 ±0.0000	0.215 ±0.0000	0.306 ±0.0000		16.343 ±0.0008
return_simple	27.678 ±0.0273	1.767 ±0.0005	0.251 ±0.0000	0.251 ±0.0000	0.195 ±0.0000	0.251 ±0.0000	0.223 ±0.0000	21.239 ±0.0161
scopes	17.854 ±0.0065	2.009 ±0.0002	0.603 ±0.0003	0.134 ±0.0000	0.124 ±0.0001	0.601 ±0.0002	0.134 ±0.0000	20.412 ±0.0014
smallfunc	46.912 ±0.0368	3.278 ±0.0002	0.251 ±0.0000	0.251 ±0.0000	0.188 ±0.0000	0.251 ±0.0000	0.251 ±0.0000	57.862 ±0.0025
sum	23.612 ±0.0201	1.440 ±0.0000	0.074 ±0.0000	0.074 ±0.0000	0.056 ±0.0000	0.074 ±0.0000	0.065 ±0.0000	31.124 ±0.0063
sum_meth	25.428 ±0.0994	1.793 ±0.0021	0.074 ±0.0000	0.074 ±0.0000		0.074 ±0.0000	0.065 ±0.0000	33.283 ±0.0360
sum_meth_attr	31.930 ±0.0046	4.351 ±0.0003	0.243 ±0.0003	0.243 ±0.0014		0.275 ±0.0001	0.220 ±0.0003	35.305 ±0.0125
total_list	8.076 ±0.0058	0.943 ±0.0003	0.363 ±0.0001	0.420 ±0.0001	0.633 ±0.0001	0.360 ±0.0002	0.246 ±0.0001	14.138 ±0.0257
walk_list	1.054 ±0.0007	0.085 ±0.0000	0.162 ±0.0001	0.208 ±0.0003	0.333 ±0.0003	0.210 ±0.0003	0.225 ±0.0001	2.218 ±0.0144
deltablue	0.901 ±0.0006	36.609 ±0.0320	0.236 ±0.0005	0.055 ±0.0002		0.246 ±0.0005	0.025 ±0.0001	7.860 ±0.1417
fannkuch	15.273 ±0.0168	2.480 ±0.0017	1.371 ±0.0004	0.742 ±0.0002	1.403 ±0.0002	1.393 ±0.0002	0.746 ±0.0002	10.676 ±0.0092
mandel		0.460 ±0.0033	0.536 ±0.0003	0.581 ±0.0002	0.287 ±0.0000	0.581 ±0.0001		4.211 ±0.0112
richards	11.901 ±0.0055	5.263 ±0.0027	0.377 ±0.0004	0.442 ±0.0002		0.392 ±0.0003	0.216 ±0.0002	10.709 ±0.0091

Making an Embedded DBMS JIT-friendly*

Carl Friedrich Bolz[1], Darya Kurilova[†2], and Laurence Tratt[3]

1 **Software Development Team, Department of Informatics, King's College London.** `http://soft-dev.org/` `http://cfbolz.de/`
2 **Institute for Software Research, School of Computer Science, Carnegie Mellon University** `http://cs.cmu.edu/~dkurilov/`
3 **Software Development Team, Department of Informatics, King's College London.** `http://soft-dev.org/` `http://tratt.net/laurie/`

Abstract

While database management systems (DBMSs) are highly optimized, interactions across the boundary between the programming language (PL) and the DBMS are costly, even for in-process embedded DBMSs. In this paper, we show that programs that interact with the popular embedded DBMS SQLite can be significantly optimized – by a factor of 3.4 in our benchmarks – by inlining across the PL / DBMS boundary. We achieved this speed-up by replacing parts of SQLite's C interpreter with RPython code and composing the resulting meta-tracing virtual machine (VM) – called *SQPyte* – with the PyPy VM. SQPyte does not compromise stand-alone SQL performance and is 2.2% faster than SQLite on the widely used TPC-H benchmark suite.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases DBMSs, JIT, performance, tracing

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.4

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.2.1.2`

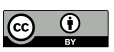
1 Introduction

Significant effort goes into optimizing database management systems (DBMSs) and programming languages (PLs), and both perform well in isolation: we can store and retrieve huge amounts of complex data; and we can perform complex computations in reasonable time. However, much less effort has gone into optimizing the interface between DBMSs and programming languages. In some cases this is not surprising. Many DBMSs run in separate processes – and often on different computers – to the PL calling them, preventing meaningful optimisation across the two. However, embedded DBMSs run in the same process as the PL calling them and are thus potentially amenable to traditional PL optimisations.

In this paper, we aim to improve the performance of PLs that call embedded DBMSs. Our fundamental hypothesis is the following:

> **Hypothesis 1** Optimisations that cross the barrier between a programming language and embedded DBMS significantly reduce the execution time of queries.

* This research was funded by the EPSRC Cooler (EP/K01790X/1) grant and Lecture (EP/L02344X/1) fellowship.
† Work performed on secondment at King's College London.



In order to test this hypothesis, we composed together PyPy and SQLite. PyPy is a widely used Python virtual machine (VM). SQLite is the most widely used embedded DBMS, shipped by default with many operating systems, and used by many applications. This composition required outfitting SQLite with a Just-In-Time (JIT) compiler, which meant that we also implicitly tested the following hypothesis:

> **Hypothesis 2** Replacing the query execution engine of a DBMS with a JIT reduces execution time of standalone SQL queries.

Thus, we tested Hypothesis 2 before testing Hypothesis 1. Our results strongly validate Hypothesis 1 but, to our initial surprise, only weakly validate Hypothesis 2.

The fundamental basis of the approach we took is to use meta-tracing JIT compilers, as implemented by the RPython system. In essence, from a description of an interpreter, RPython derives a VM with a JIT compiler. PyPy is an existing RPython VM for the Python language. SQLite, in contrast, is a traditional interpreter implemented in C. We therefore ported selected parts of SQLite's core opcode dispatcher from C to RPython, turning SQLite into a (partially) meta-tracing DBMS. While we left most of the core DBMS parts of SQLite (e.g. B-tree manipulation, file handling, and sorting) in C, we refer to our modified research system as *SQPyte* to simplify our exposition.

Relative to SQLite, SQPyte is 2.2% faster on the industry standard TPC-H benchmark suite [28]. We added specific optimisations intended to exploit the fact that SQLite is dynamically typed, but, as this relatively paltry performance improvement suggests, to little effect. We suspect that much more of SQLite's C code would need to be ported to RPython for this figure to significantly improve.

Since TPC-H measures SQL query performance in isolation from a PL, we then created a series of micro-benchmarks which measure the performance of programs which cross the PL / DBMS boundary. SQPyte is 3.4× faster than SQLite on these micro-benchmarks, showing the benefits of being able to inline from PyPy into SQPyte.

The major parts of this paper are as follows. After describing how SQPyte was created from SQLite (Section 3), we test Hypothesis 2 (Section 5). We then describe how PyPy and SQLite are composed together (Section 6) allowing us to test Hypothesis 1 (Section 7).

SQPyte's source code, and all benchmarks used in this paper, can be downloaded from `http://dx.doi.org/10.4230/DARTS.2.1.2`.

2 Background

After briefly defining the difference between external and embedded databases, this section summarizes the relevant aspects of SQLite and meta-tracing for those readers who are unfamiliar with them. Note that this paper deals with several different technologies, each of which uses slightly different terminology. We have deliberately imposed consistent terminology in our discussions to aid readers of this paper.

2.1 Embedded DBMSs

From the perspective of this paper, DBMSs come in two major variants.

External DBMSs are typically used for large quantities of vital data. They run as separate processes and interactions with them require inter-process calls (IPCs) or network communications. The overhead of IPC varies depending on operating system and hardware, but translating a function that returns a simple integer into an IPC equivalent typically leads to a slowdown of at least 5 orders of magnitude. Since there is a fixed cost for each

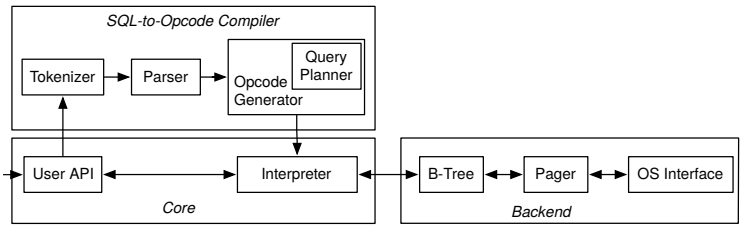


Figure 1 SQLite architecture.

call, unrelated to the quantity of data, small repeated IPC calls are costly. Programmers thus use various techniques to bunch queries together to lower the fixed cost overhead of IPC. When bunching is impossible, it is not unusual for IPC costs to dominate interaction with an external DBMS. This effect is even more pronounced for databases which run over a network.

Embedded DBMSs are typically used for smaller quantities of data, often as part of a desktop or mobile application. They run within the same memory space as a user program, removing IPC costs. However, embedded DBMSs tend to be used as pre-packaged external libraries, meaning that there is no support for optimising calls from the user application to the embedded DBMS.

2.2 SQLite

SQLite[1] is an embedded DBMS implemented as a C library. It is the most commonly used embedded DBMS, installed as standard on operating systems such as OS X, and widely used by desktop and mobile applications (e.g. email clients).

Figure 1 shows SQLite's high-level architecture: its core provides the user-facing API that external programs use, as well as an interpreter for running queries; the backend stores and retrieves data in memory and on disk; and the compiler translates SQL into an instruction sequence. Instructions consist of an opcode (i.e. the 'type' of the instruction) and up to five operands p1...p5 (p1, p2, and p3 are always 32-bit integers; p4 is of variable size; and p5 is an unsigned character), which are variously used to refer to registers, program counter offsets, and the like. SQLite is dynamically typed and SQL values are either Unicode strings, arbitrary binary 'blobs', 64-bit numbers (integers or floating point), or null. SQL values are stored in a single C-level type `Mem`, which can also store other SQLite internal values (e.g. row sets). The opcode dispatcher[2] contains an arbitrary number of registers (each of which stores a `Mem` instance), and zero or more cursors (pointers into a table or index).

Figure 2 shows an elided version of the `Mem` struct, which plays a significant role in SQLite and therefore in much of our work. `flags` is a bit field that encodes the type(s) of the values stored in the struct. Most SQL values and all SQLite internal values are stored in the `MemValue` union. Strings are stored on the heap with a pointer to them in `z`. In some cases, `Mem` can store two SQL values simultaneously. For example, an integer cast at run-time to a string will store the integer value in `i` and the string in `z`, so that subsequent casts have zero cost. In such cases, `flags` records that the value has more than one run-time type.

[1] `http://www.sqlite.org/`
[2] SQLite refers to this as its 'virtual machine', but we reserve that term for other uses.

```
struct Mem {
  union MemValue {
    double r;
    i64 i;
    ...
  } u;
  u16 flags;
  char *z;
  ...
};
```

Figure 2 An elided view of SQLite's `Mem` struct, used to represent SQL values.

2.3 Meta-tracing

Tracing is a technique for writing JIT compilers that record 'hot loops' in a program and convert them to machine code. Traditionally, tracing requires manually creating both an interpreter and a trace compiler (see [1, 8]). In contrast, meta-tracing takes an interpreter as input and from it automatically creates a VM with a tracing JIT compiler [22, 27, 4, 30, 3]. At run-time, user programs begin their execution in the interpreter. When a hot loop in the user program is encountered, the actions of the interpreter are traced, optimized, and converted to machine code. Since the initial traces are voluminous, the trace optimiser is often able to reduce them to a fraction of their original size, before they are converted to machine code. Subsequent executions of the loop then use the fast machine code version rather than the slow interpreter. Guards are left behind in the machine code so that execution paths that stray from the trace revert back to the interpreter.

In this paper we use RPython, the major extant meta-tracing language. RPython is a statically typeable subset of Python with a type system similar to that of Java, garbage collection, and high-level data types (e.g. lists and dictionaries). Despite this, VMs written in RPython have performance levels far exceeding traditional interpreter-only implementations [5]. The specific details of RPython are generally unimportant in most of this paper, and we do not concentrate on them: we believe that one could substitute any reasonable meta-tracing language (or its cousin approach, self-optimizing interpreters with dynamic partial evaluation [29]) and achieve similar results.

3 SQPyte

In order to test Hypothesis 2, we created a variant of SQLite called SQPyte, where parts of SQLite's interpreter are ported from C into RPython. SQPyte is therefore meta-tracing compatible, meaning that SQL queries which use the RPython parts of SQPyte's interpreter are JIT compiled. In the rest of this section we explain the details of our porting.

It is important to note that SQPyte is not a complete, or even a majority, rewrite of SQLite. Fortunately for us, RPython is compiled into C by default, which makes mixing RPython and C code simple, with zero overhead for common operations such as function calls. This means that we were able to leave most of SQLite in C, calling such code as necessary, and only porting the minimum possible to RPython.

However, only code which is written in RPython can be JIT compiled: calls to C cannot be inlined by the meta-tracer, reducing the possibilities for optimisations. We therefore worked incrementally, porting code to RPython only after we had recognized that it was on the critical performance path and likely to benefit from meta-tracing. Note that we are not suggesting that SQPyte would not benefit from having more code in RPython, simply that

```
1  import sqpyte
2  conn = sqpyte.Connection("tpch.db")
3  sum_qty = 0
4  sum_base_price = 0
5  sum_disc_price = 0
6  iterator = conn.execute("SELECT quantity, extendedprice, discount FROM lineitem")
7  for quantity, extendedprice, discount in iterator:
8      sum_qty += quantity
9      sum_base_price += extendedprice
10     sum_disc_price += extendedprice * (1 - discount)
```

Figure 3 An example use of the `sqpyte` module in PyPy. This example program connects to the `tpch.db` database and computes the total quantity, the sum of the base price, and the sum of the discounted price of all items. balance of all accounts.

with our available effort levels, we had to focus our attention on those parts of SQLite that we believed were most relevant. As a rough indication of size, we ported 1550 lines of C code and wrote 1300 lines of RPython code to replace it.

In this section, we first introduce this paper's running example, before explaining how SQPyte was created from SQLite.

3.1 Running Example

SQPyte adds a module called `sqpyte` to PyPy. This module exposes a standard Python DBMS API[3] that allows Python programs to directly interact with SQPyte. Although it is mostly irrelevant from this paper's perspective, the `sqpyte` module's interface is a strict subset of that exposed by the `sqlite3` module, which is shipped as standard with PyPy and other Python implementations.

Figure 3 shows the running example we use throughout this paper. After connecting to a database (line 2), the example starts the execution of an SQL query (line 6), receiving an iterator object in return. As the iterator is pumped for new values in the `for` loop (line 7), SQPyte lazily computes further values. Each iteration yields 3 SQL values that are processed by regular Python code (lines 8–10).

3.2 Opcodes

SQLite's interpreter executes instructions until either a query is complete, a new row of results is produced, or an error occurs. Each iteration of the interpreter loop loads the instruction at the current program counter and jumps to an implementation of the instruction's opcode.

The first stage of the SQPyte port was to port the opcode dispatcher from C to RPython, as shown in Figure 4. This can be thought of as having three phases. First, since we wanted to reuse some of SQLite's opcode's implementations, we split them out from the (rather large) switch statement they were part of into individual functions (one per opcode). Second, we translated the main opcode dispatcher loop itself. Finally, we added the two annotations[4] required by RPython to make SQPyte's interpreter meta-tracing compatible. These annotations inform the meta-tracing system about the current execution point of the

[3] The API is defined in `https://www.python.org/dev/peps/pep-0249/`
[4] While these are written using normal function call syntax, they are treated specially by RPython.

```
SQLITE_PRIVATE int sqlite3VdbeExec(
  Vdbe *p) {
  int pc=0;
  Op *aOp = p->aOp;
  Op *pOp;
  int rc = SQLITE_OK;
  ...
  for(pc=p->pc; rc==SQLITE_OK; pc++){
    ...
    switch( pOp->opcode ){
      case OP_Goto: {
        pc = pOp->p2 - 1;
        ...
        break;
      }
      case OP_Gosub: {
        ...
        break;
      }
      case ...: { ... }
    }
  }
}
```

```
def mainloop(self):
  rc = CConfig.SQLITE_OK
  pc = self.p.pc
  while True:
    jitdriver.jit_merge_point(pc)
    if rc != CConfig.SQLITE_OK:
      break
    op = self._hlops[pc]
    opcode = op.get_opcode()
    oldpc = pc
    if opcode == CConfig.OP_Goto:
      pc, rc =
         self.python_OP_Goto(pc, rc, op)
    elif opcode == CConfig.OP_Gosub:
      pc = self.python_OP_Gosub(pc, op)
    elif ...:
      ...
    pc += 1
    if pc <= oldpc:
      jitdriver.can_enter_jit(pc)
```

Figure 4 An elided version of the opcode dispatcher, with the original C on the left and the ported RPython on the right. The RPython interpreter requires the `jit_merge_point` and `can_enter_jit` annotations to enable the meta-tracing system to identify hot loops.

system (for example, the program counter and the known types of the registers) so that it can determine if JIT compilation or execution can, or should, occur. `can_enter_jit` is called when a loop is encountered: if that happens often enough, then tracing of the loop occurs (i.e. the loop is, ultimately, converted into machine code). `jit_merge_point` allows the meta-tracing system to determine whether there is a machine code version of the current execution point, or whether the interpreter must be used instead.

Figure 5 shows an example of an opcode in SQLite and its SQPyte port. As this example suggests, many aspects of the porting process are fairly obvious, though some are slightly obscured by the greater use of helper functions in RPython (these make the RPython version easier to understand in isolation, but can make C-to-RPython comparisons a little harder). To ensure that we are able to make an apples-to-apples performance comparison, we ported all aspects of SQLite's C code enabled in the single-threaded default build. This meant that we did not need to port parts such as the `assert` and `VdbeBranchTaken` macros (a complete list of unported aspects can be found in Appendix A), which are no-ops in the default build and thus have no run-time effect whatsoever.

SQLite's opcode dispatcher contains several `goto`s to deal with exceptional situations, as can be seen in Figure 6. Since SQPyte breaks opcodes into different functions, this behaviour is no longer tenable, since we can't `goto` across different functions.[5] We thus ported labelled blocks to explicit functions, and `goto` jumps to function calls, with each followed by a `return`. This achieves the same overall program flow at the cost, when interpreting, of requiring more function calls and, at any given time, an extra stack frame.

Porting all of SQLite's opcodes to RPython would be a significant task, and not necessarily a fruitful one—some opcodes are called rarely, and some would benefit little from

[5] Not, it should be added, that RPython has a `goto` construct.

```
1 case OP_IfPos: {
2   pIn1 = &aMem[pOp->p1];
3   assert(pIn1->flags&MEM_Int);
4   VdbeBranchTaken(pIn1->u.i > 0, 2);
5   if (pIn1->u.i > 0) {
6      pc = pOp->p2 - 1;
7   }
8   break;
9 }
```

```
1 def python_OP_IfPos(hlquery, pc, op):
2     pIn1 = op.mem_of_p(1)
3     if pIn1.get_u_i() > 0:
4         pc = op.p2as_pc()
5     return pc
```

Figure 5 An example port of an opcode from C to RPython. The `IfPos` opcode is a conditional jump: it loads the register specified by its p1 operand (lines 2 in C and RPython) and compares the resulting value (as an integer) to 0 (line 5 in C; line 3 in RPython). If the value is greater than zero it jumps to the position specified by the p2 operand (line 6 in C; line 4 in RPython). As this example shows, the RPython code makes greater use of helper functions and removes functions that do not appear in the production version of SQLite (both `assert` and `VdbeBranchTaken` are no-ops in production builds).

```
1  case OP_MakeRecord: {
2    ...
3    if (...)
4      goto no_mem;
5    ...
6  }
7  case OP_Yield: {
8    ...
9  }
10 ...
11 no_mem:
12   ...
```

```
1  def OP_MakeRecord(...):
2    ...
3    if ...:
4      return hlquery.gotoNoMem(pc)
5    ...
6  def OP_Yield(...):
7    ...
8  ...
9  def gotoNoMem(hlquery, pc):
10   ...
```

Figure 6 An example of how we port `goto`s in an opcode into SQPyte. We ported 4 `goto` labels, making each a separate function (e.g. `gotoNoMem`). Instead of executing a `goto`, SQPyte calls the appropriate function, and then immediately returns to the main interpreter loop, thus mimicking the control flow of SQLite.

meta-tracing. We thus chose to focus our porting efforts on those opcodes which we believed would see the greatest benefit from meta-tracing (chiefly those which change the program counter, or manipulate type flags). Of SQLite's 153 opcodes, we ported 61 into RPython. A further 42 opcodes were needed by queries we support, but we judged that they were unlikely to benefit from JIT optimisations (because, for example, they immediately call SQLite's B-tree manipulating functions, which remain in C and are thus opaque to the meta-tracer). We thus copied these opcodes directly from SQLite, leaving them in C. Since we removed the giant `switch` statement these C opcodes were originally part of, each was put into its own function, mirroring those opcodes ported to RPython. Since this is a tedious, mechanical task, we copied only those opcodes we needed: 50 of SQLite's opcodes are thus currently unsupported by SQPyte, and an exception is raised if a query tries to use one of them.

3.3 Optimizing the `flags` Attribute

Most SQLite opcodes read or write to registers, each of which contains a `Mem` struct. Typically, such opcodes must first read the `flags` attribute of the `Mem` struct to determine what type of value is stored within it. Many opcodes also write to this flag when storing a result. SQLite is completely dynamically typed – different entries in a database column, for example, may be of different types – and, in essence, the `flags` attribute is how the dy-

```
case OP_NotNull: {
  pIn1 = &aMem[pOp->p1];
  VdbeBranchTaken((pIn1->flags &
                   MEM_Null) == 0, 2);
  if( (pIn1->flags & MEM_Null)==0 ){
    pc = pOp->p2 - 1;
  }
  break;
}
```

```
def OP_NotNull(hlquery, pc, op):
    pIn1, flags1 = op.mem_and_flags_of_p(1)
    if flags1 & CConfig.MEM_Null == 0:
        pc = op.p2as_pc()
    return pc
```

Figure 7 An example of porting operations on the `flags` attribute from C to RPython. In this case, the `NotNull` opcode jumps to a different `pc` if the register indexed by the opcode's p1 operand is not `Null`. The `op.mem_and_flags_of_p(1)` helper function reads the register specified by the `p1` argument and returns the appropriate `Mem` structure and its flags.

```
@cache_safe(mutates="p2")
def python_OP_String(self, op):
  capi.impl_OP_String(...)
```

Figure 8 An example of the side-effect annotation used to specify which `flags` attributes a C opcode can invalidate. In this case, the annotation specifies that the `OP_String` opcode invalidates the entry for the register specified by the opcode's `p2` argument.

namic types are encoded. However, dynamically typed languages tend to be surprisingly type-constant at run-time, which is why JIT compilers are effective on such languages [5]. A reasonable expectation is thus that, as with other dynamically typed languages, most SQL queries are fairly type-constant. We thus made the following hypothesis:

> **Hypothesis 3** Exposing the type information in the `flags` attribute associated with registers allows the JIT compiler to speed up query execution.

We addressed this hypothesis by adding a mechanism to SQPyte that allows the trace optimiser to reason about the `flags` attributes in registers' `Mem` structs. This is implemented as a cache storing known `flags` values (in essence, a close cousin of Self-style maps [7]). When an opcode reads the `flags` attribute from a `Mem` struct in a register, the trace records the read; SQPyte is annotated such that the trace optimizer can remove all the subsequent reads of the `flags` attribute of the same register, using the previously read value. Similarly, subsequent reads are optimised away after a `flags` attribute is written to. While the trace optimiser is normally able to perform redundant load optimisations such as this automatically, it is unable to reason about the `flags` attribute, which is stored in a (semi-opaque) C object, hence our need to manually help the trace optimiser.

Figure 7 shows an example of the `NotNull` opcode which operates on the `flags` attribute. The RPython method `mem_and_flags_of_p()` is the heart of the `flags` optimisation. If this opcode is part of a trace which has earlier read `p1`'s flags, and there are no intermediate writes, then the call to `mem_and_flags_of_p(1)` will be entirely removed by the trace optimizer.

Those opcodes which remain in C have their RPython wrapping function annotated with side-effect information [19] to specify which registers' flags may have been changed by the opcode. After the opcode has been executed, the tracer knows that any previous information about the `flags` fields of the relevant registers is now invalid. An example annotation is shown in Figure 8.

Two opcodes are handled somewhat specially. First is SQLite's most frequently executed opcode, `Column`, which reads one value from a row. This relatively complex opcode analyses

```
static void sin_sqlite(
   sqlite3_context *context, int argc,
   sqlite3_value **argv) {
  double value =
    sqlite3_value_double(argv[0]);
  double result = sin_sqlite(value);
  sqlite3_result_double(context, result);
}
...
sqlite3_create_function(db, "sin", 1,
     SQLITE_UTF8, NULL, &sin_sqlite,
     NULL, NULL)
```

```
def sin(func, args, result):
    arg = args[0].sqlite3_value_double()
    result.sqlite3_result_double(
        math.sin(arg))
...
db.create_function("sin", 1, sin)
```

Figure 9 An example of registering a `sin` function with SQLite (C) and SQPyte (RPython). Note that both APIs require specifying the name of the function, the number of parameters, and a pointer to its implementation.

the packed B-Tree representing a row, extracts the requested column, and stores it into the register specified by the `p3` operand. Because most of this opcode calls out to DBMS C code, translating the entire (rather large) opcode to RPython would be a tedious exercise. Since the opcode can change register `p3`'s flags, this meant that most calls to this opcode were followed by a check of `p3`'s flags—including a read from memory. We removed these reads by having the `Column` opcode return both the return code of the opcode and the most recent value of `p3`'s `flags` encoded into one number. The trace optimizer is then able to use the returned value to determine if its knowledge of `p3`'s flags is current or not, without having to read from memory.

We also optimized the `MakeRecord` opcode to expose `flags` information to the JIT compiler. This opcode reads from a specified number of n registers and produces a packed representation of the content of these registers, used for later storage, and placed in the register specified by `p3`. Since n is constant for each specific call of the opcode, we marked `MakeRecord`'s inner loop as unrollable, so that the resulting trace contains separate code for each register read. As well as removing the general loop overhead, this allows the trace optimizer to reason about the `flags` operations involved in reading from each of the n registers.

3.4 SQL Functions and Aggregates

SQLite has both regular functions (henceforth simply 'functions') and aggregates.[6] Both take a number of arguments as input. Functions produce a single result per row that they are applied to, whereas aggregates (e.g. `max`) reduce many rows to a single value.

SQLite implements functions and aggregates in C, but does not hard-code them into the interpreter: each is registered via an API to the SQL interpreter. If SQPyte kept these functions in C, then the meta-tracer would have to treat them as opaque calls, preventing inlining. Fortunately, we were able to easily add an RPython mirror of SQLite's C interface for registering functions and aggregates. Figure 9 shows an example of the two interfaces alongside each other. Aggregates are implemented in similar manner, albeit in two parts: a step function (e.g. an acumulator) and a finalizer function (e.g. a divisor). We implemented a small number of commonly called SQL aggregates in RPython: `sum`, `avg`, and `count`.

To enable inlining, we also had to alter the opcodes which call functions and aggregates.

[6] There are also user-defined collation functions which we did not optimize in a special way.

```
0|Init|0|12|0||00|
1|OpenRead|0|8|0|7|00|
2|Rewind|0|10|0||00|
3|Column|0|4|1||00|
4|Column|0|5|2||00|
5|RealAffinity|2|0|0||00|
6|Column|0|6|3||00|
7|RealAffinity|3|0|0||00|
8|ResultRow|1|3|0||00|
9|Next|0|3|0||01|
10|Close|0|0|0||00|
11|Halt|0|0|0||00|
12|Transaction|0|0|23|0|01|
13|TableLock|0|8|0|LineItem|00|
14|Goto|0|1|0||00|
```

```
# SQLite opcode Next
...
i168 = call(sqlite3BtreeNext, ...)
guard_value(i168, 0)
# SQLite opcode Column
i173 = call(impl_OP_Column, 3, ...)
guard_value(i173, 262144)
# SQLite opcode Column
i174 = call(impl_OP_Column, 4, ...)
guard_value(i174, 524288)
# SQLite opcode RealAffinity
# SQLite opcode Column
i175 = call(impl_OP_Column, 6, ...)
guard_value(i175, 524288)
# SQLite opcode RealAffinity
# SQLite opcode ResultRow
...
i178 = call(sqlite3VdbeCloseStatement, ...)
i179 = int_is_true(i178)
guard_false(i179)
...
```

Figure 10 On the left, SQLite's rendering of the opcodes generated for the query `SELECT quantity, extendedprice, discount FROM lineitem`. The first column represents the program counter; the second column the opcode; and the remaining columns the operands to the opcode. On the right, an elided SQPyte optimized trace for one result row of the query. Note that after optimisation, some opcodes have no operations in the trace.

The `Function` opcode is responsible for calling functions and is easily altered to permit inlining into RPython functions. Calling an aggregate uses two opcodes: `AggStep` initializes the aggregator, and calls the step function on each row; and `AggFinalize` returns the final aggregate result.

3.5 Overflow checking

An advantage of controlling assembler code generation in a JIT is that one can make use of machine code features that are hard to express directly in C. RPython uses this to allow for overflow check's on arithmetic operations to be performed without checking the operations concrete result (i.e. it makes use of hardware features which few programming languages directly expose). We make use of this feature in the implementation of arithmetic opcodes such as `Add`, `Sub`, and `Mul`. If results overflow an integer, each of these switch to a floating point representation.

3.6 From Query to Trace

We now recall the SQL query used in the running example of Figure 3: `SELECT quantity, extendedprice, discount FROM lineitem`. SQLite's compiler translates this into a sequence of opcodes, which can be seen in Figure 10.

The high-level structure of the query opcode as as follows. The query starts by calling `Init` (opcode 0) which sets the program counter to its second operand, in this case 12. This creates a new transaction (opcode 12), locks the table (opcode 13) before jumping (opcode 14) to the main loop query.

The main loop operates on every row in the database (opcodes 3–9). The `Column` opcodes (opcodes 3, 4, and 6) read values from the `quantity`, `extendedprice`, and `discount` columns in a row respectively. Although SQLite attaches type information to columns, these are, in

a sense, optional: any given value within a column may be of an arbitrary type. Thus the `RealAffinity` opcodes (opcodes 5 and 7) inspect the `extendedprice` and `discount Mem` structs: if they hold floats (which SQLite terms 'reals'), the result is a no-op; if they hold integers, then they are cast to floats. The `ResultRow` opcode (opcode 8) returns n results (registers p1...p1+p2-1 i.e. 1, 2, and 3 in our example) to the caller, suspending query execution. Upon resumption, the `Next` opcode (opcode 9) advances the database cursor to the next row in the table and updates the program counter to its second operand – in this case 3. If there is no further data in the table, execution continues to the next opcode, which closes the database connection (opcode 10) before halting query execution (opcode 11).

If the heart of the query opcode is in a hot loop traced by SQPyte's tracing JIT compiler, then the result is as in Figure 10. Traces always start with the `Next` opcode, since the iteration that triggered the tracing threshold was suspended as part of that opcode and thus the next iteration starts when the query is resumed. `Next` calls the `sqlite3BtreeNext` C function, which advances the database row (line 3), with a guard ensuring the result is 0, which indicates success (line 4). The `Column` opcodes also call a C function, but the return type is more complex, encoding both the function's error code and the flags of the register that `Column` stored a result into. Assuming the guard holds, the remainder of the trace thus implicitly knows the type of the register in question (see Section 3.3). This allows the trace optimizer to remove the dynamic checks of the `RealAffinity` opcode all together. As this shows, the trace optimizer is often able to remove a substantial portion of the operations in an SQPyte trace.

4 Experimental methodology

We have two distinct experimental sections (primarily addressing, in order, Hypotheses 2 and 1), both sharing a common methodology. First we compare SQPyte to SQLite and to H2, a widely used embedded Java database. H2 is of most interest to Hypothesis 1, where it allows us to understand how SQPyte and PyPy's cross-system inlining in RPython compares to Java and H2's cross-system inlining on HotSpot. However, to put H2's cross-system performance into perspective, it is also useful to see its performance on queries that address Hypothesis 2. SQPyte is based on SQLite 3.8.8.2. We used PyPy 5.0 and H2 1.4.191.

In both experimental sections, we run a number of queries. Each query is run in 5 fresh processes; each process runs 50 iterations of the query. We placed a 1 hour timeout on each process. We report the mean and 99% confidence intervals of all iterations across all processes (i.e. 250 in total). Note that by including all iterations, we are implicitly including those where the VMs may be warming up.

As recommended by its documentation, SQLite was configured in single-threaded mode, as was SQPyte. We used H2 in its default configuration. All benchmarks were run on an otherwise idle Intel i7-4790 machine, with 32 GiB RAM, and Debian 8.1. We turned off hyper-threading and turbo boost in the BIOS: hyper-threading is of little use to our single-threaded benchmarks, and adds noise to measurements; and turbo boost's benefits disappear as soon as the CPU gets too hot, ruining benchmarking. The database files were put into a RAM disk to ensure that possible data caching effects between DBMSs were reduced. We performed an initial run of our experiment to ensure that it never caused the machine to swap memory to disk.

Table 1 SQPyte, SQLite, and H2 performance on the TPC-H benchmark set. For each query, the first row shows the absolute time in seconds; the second row shows the performance relative to SQPyte as a factor. Queries where SQLite or H2 are faster than SQPyte are shown in bold. Queries where SQLite or H2 are, within the confidence interval, equivalent in performance to SQPyte are shown in grey. Note that Query 19 timed out on H2, hence the lack of data.

Benchmark	SQPyte	SQLite	H2
Query 1 (s)	6.929 ± 0.0352	8.715 ± 0.0083	13.168 ± 0.1584
×		1.258 ± 0.0065	1.901 ± 0.0254
Query 2 (s)	0.298 ± 0.0098	0.305 ± 0.0024	12.890 ± 0.0787
×		1.025 ± 0.0340	43.324 ± 1.4273
Query 3 (s)	2.933 ± 0.0329	3.098 ± 0.0100	10.636 ± 0.0490
×		1.056 ± 0.0122	3.626 ± 0.0452
Query 4 (s)	0.345 ± 0.0038	0.345 ± 0.0014	2.243 ± 0.0265
×		0.998 ± 0.0121	6.494 ± 0.1081
Query 5 (s)	1.111 ± 0.0145	1.116 ± 0.0239	158.297 ± 0.5371
×		1.004 ± 0.0261	142.473 ± 1.9971
Query 6 (s)	0.701 ± 0.0081	0.794 ± 0.0040	9.197 ± 0.0571
×		1.134 ± 0.0147	13.125 ± 0.1741
Query 7 (s)	2.630 ± 0.0070	2.847 ± 0.0318	116.322 ± 0.3302
×		1.083 ± 0.0126	44.236 ± 0.1710
Query 8 (s)	2.510 ± 0.0141	2.519 ± 0.0646	161.185 ± 0.9576
×		1.003 ± 0.0265	64.225 ± 0.5471
Query 9 (s)	10.062 ± 0.0448	10.269 ± 0.0276	121.319 ± 0.9515
×		1.021 ± 0.0055	12.055 ± 0.1137
Query 10 (s)	0.019 ± 0.0056	**0.009 ± 0.0006**	17.082 ± 0.0660
×		**0.499 ± 0.1632**	918.900 ± 292.4240
Query 11 (s)	0.604 ± 0.0071	0.647 ± 0.0026	**0.494 ± 0.0174**
×		1.071 ± 0.0134	**0.819 ± 0.0312**
Query 12 (s)	0.938 ± 0.0062	1.027 ± 0.0013	20.129 ± 0.0604
×		1.094 ± 0.0073	21.455 ± 0.1571
Query 13 (s)	2.721 ± 0.0135	2.818 ± 0.0123	14.350 ± 0.0840
×		1.036 ± 0.0072	5.274 ± 0.0427
Query 14 (s)	0.792 ± 0.0102	0.863 ± 0.0043	65.708 ± 0.3407
×		1.090 ± 0.0156	82.944 ± 1.1772
Query 15 (s)	20.636 ± 0.2254	20.881 ± 0.5542	**0.009 ± 0.0036**
×		1.011 ± 0.0300	**0.000 ± 0.0002**
Query 16 (s)	0.410 ± 0.0074	0.447 ± 0.0013	0.583 ± 0.0155
×		1.089 ± 0.0199	1.420 ± 0.0459
Query 17 (s)	0.107 ± 0.0008	0.114 ± 0.0001	0.516 ± 0.0141
×		1.067 ± 0.0082	4.805 ± 0.1354
Query 18 (s)	2.449 ± 0.0144	2.822 ± 0.0351	15.210 ± 0.0745
×		1.152 ± 0.0164	6.211 ± 0.0492
Query 19 (s)	8.140 ± 0.1333	8.114 ± 0.0397	—
×		0.997 ± 0.0169	—
Query 20 (s)	80.386 ± 0.2692	81.378 ± 0.2668	**10.210 ± 0.0450**
×		1.012 ± 0.0049	**0.127 ± 0.0007**
Query 21 (s)	8.661 ± 0.0347	9.066 ± 0.1017	**8.146 ± 0.0651**
×		1.047 ± 0.0124	**0.941 ± 0.0086**
Query 22 (s)	0.087 ± 0.0036	0.087 ± 0.0003	1.728 ± 0.0215
×		1.003 ± 0.0408	19.830 ± 0.8475
Geometric mean ×		1.022 ± 0.0151	6.172 ± 0.1484

5 Testing Hypothesis 2: SQPyte using TPC-H

To evaluate Hypothesis 2 – in essence, does SQPyte have better performance than SQLite when both are used standalone? – we measure SQPyte's performance on the widely used TPC-H benchmark set [28]. TPC-H's 22 queries utilise 8 tables, which can be populated with different quantities of data: we chose the 1.5GiB variant, which contains 8.7 million rows. Table 1 shows the resulting comparison of the 3 DBMSs.

Overall, SQLite is $2.2 \pm 1.53\%$ slower than SQPyte. This validates Hypothesis 2, though only weakly. A more detailed look at the data reveals a slightly muddy story. All but 1 query is faster in SQPyte than SQLite, with a maximum improvement over SQLite of $25.8 \pm 0.65\%$ faster (query 1). Query 10 is the outlier, with SQPyte a little over 100% slower than SQLite. This is simply because the query executes two orders of magnitude more quickly than all but one other query ($0.0093 \pm 0.00061s$). SQPyte's performance is thus dominated by the time the JIT takes to produce machine code while in the first iteration of the benchmark. However, even if query 10 were removed from the results, the overall speedup would only be $5.8 \pm 0.45\%$ — substantially better, but still somewhat weak validation of Hypothesis 2. These results strongly suggest that for benchmarks such as TPC-H, SQLite and SQPyte's overall performance is dominated not by the interpreter but by the core DBMS (e.g. operations on B-trees). Porting more of SQLite to RPython may improve performance further, but it is hard to estimate the likely gains, and the effort involved would be significant.

H2 is, on average, $6.172 \pm 0.1476\times$ significantly slower than both SQPyte and SQLite. Query 19 exceeded our one hour timeout. Query 15, on the other hand, is almost 3 orders of magnitude faster than SQLite and SQPyte. The reason for that is that Query 15 uses an SQL view, which H2 is able to cache, but which SQLite continually, and unnecessarily, recomputes (a well known SQLite issue).

6 Composing SQPyte and PyPy

As with most embedded DBMSs, SQLite is rarely used standalone. Instead, a user program interacts with SQLite through a language-specific library, as shown in Figure 3. Thus the overall performance experienced by the user is dictated by 3 factors: the performance of the programming language the user program is implemented in; the performance of the embedded DBMS; and the performance of interactions across the PL / DBMS boundary. Hypothesis 1 captures our intuition that substantial optimisations are possible if one can optimize across the PL / DBMS boundary.

In order to test Hypothesis 1, we composed together SQPyte and PyPy. PyPy is an industrial strength meta-tracing Python VM, which can be used as a drop-in replacement for the standard Python interpreter. Since PyPy is written in RPython, we were able to extend SQPyte and PyPy so that tracing can bridge across the PL / DBMS boundary. Put another way, database calls from PyPy inline code in SQPyte's RPython interpreter.

The major part of the composition is the `sqpyte` module added to PyPy, which allows programs run under PyPy to execute queries in SQPyte (see Section 3.1 for the user-facing details about this module). Since it is written in RPython, `sqpyte` simply imports SQPyte as another RPython module. Simple queries thus inline across the interface without significant effort, with all the normal benefits of trace optimisation. The optimisation of SQPyte's `flags` attribute (see Section 3.3) means that in many cases data moved between SQPyte and PyPy requires neither an explicit conversion nor even a guard. Some queries can't be

```
1  label(i144, f147, f154, i55, f57, f59, ...)
2  # for quantity, extendedprice, discount in iterator:
3  ...
4  i161 = <MemValue 87403720>.u.i
5  f162 = <MemValue 87403776>.u.r
6  f163 = <MemValue 87403832>.u.r
7  ...
8  # At this point, there is a copy of the trace from Figure 10
9  ...
10 # sum_qty += quantity
11 i186 = int_add_ovf(i144, i161)
12 guard_no_overflow()
13
14 # sum_base_price += extendedprice
15 f188 = float_add(f147, f162)
16
17 # sum_disc_price += extendedprice * (1 - discount)
18 f189 = float_sub(1.000000, f163)
19 f190 = float_mul(f162, f189)
20 f191 = float_add(f154, f190)
21 ...
22 jump(i186, f188, f191, i161, f162, f163, ...)
```

Figure 11 An elided version of the optimized trace of the Python program and SQL query from Figure 3, annotated to explain which parts relate to which parts of the input program. Notice that we have removed a significant part of the trace at line 8, since it is identical to that found in Figure 10. As a rough gauge, the complete unoptimized trace contains 375 operations; the optimized trace contains 137 operations.

sensibly inlined, notably those which induce a loop in SQPyte's interpreter such as SQL joins. In such cases, PyPy and SQLite optimize their traces independently of each other.

Using the running example from Figure 3, the resulting trace in our composition can be seen in Figure 11. The optimized trace starts by reading the integer and two float values (`quantity`, `discount`, and `lineitem` respectively) from the `Mem` structures of the most recently read row (lines 4–6). Next is a structurally identical clone (with only α-renamed SSA variables) of the trace from Figure 10 (see the explanation in Section 3.6), which establishes the datatypes of the three fetched values. The remainder of the trace (lines 10–22) correspond to the Python `for` loop in Figure 3. Since the low-level integer and float datatypes used by SQPyte and PyPy are the same, there is no need to convert between the two, and, for example, the SQPyte integer (line 5) can be used as-is in the PyPy part of the trace (line 11). Indeed, with the exception of the overflow guard imposed by Python (line 12), the optimized trace melds SQPyte and PyPy together such that it is difficult to distinguish the two.

6.1 Calling back from SQPyte to Python

SQLite allows callbacks during an SQL query to functions in the calling PL. For example, an end user can register a new aggregate, which consumes a sequence of SQL rows and returns a value as shown in Figure 12. In our context, Python can call SQLite, which calls Python, which returns to SQLite, and which finally returns to Python. Since SQLite is reentrant, this pattern of nesting can be arbitrarily deep.

While the ability to register such callbacks is powerful, it means that data and control flow pass over the programming language / DBMS boundary much more frequently than normal. The `sqpyte` module not only supports callback of regular functions and aggregates,

```
1  class MySum(object):
2      def __init__(self):
3          self.sum = 0
4
5      def step(self, x):
6          self.sum += x
7
8      def finalize(self):
9          return self.sum
10 conn.create_aggregate("mysum", 1, MySum)
```

Figure 12 A pure Python implementation of a `sum` aggregate, registered using `sqpyte`'s public API (line 10). Put another way, this example is not part of SQPyte's RPython system, and is normal end-user code. For every row of the query, the `step` method is called. The aggregation's result is computed by calling the `finalize` method.

```
1  d = {}
2  for key, suppkey in conn.execute("""SELECT PartSupp.PartKey, PartSupp.SuppKey
3                                      FROM PartSupp;"""):
4      cursor = conn.execute("SELECT Part.name FROM Part WHERE part.PartKey = ?;", [key])
5      partname, = cursor.next()
6      cursor = conn.execute("""SELECT Supplier.name FROM Supplier
7                               WHERE Supplier.SuppKey = ?;""", [suppkey])
8      suppname, = cursor.next()
9      d[partname] = suppname
10 return d
```

Figure 13 The core of the `pythonjoin` micro-benchmark.

but enables inlining whenever possible. Enabling this meant that we had to convert a few more parts of SQLite into RPython, so that the full path from Python to SQPyte back to Python is in RPython. Much as we did when calling SQPyte from Python, we make use of tracings natural tendency to inline; though, as before, Python callbacks which have loops lead to separate traces on either side.

7 Evaluation of Hypothesis 1: SQPyte and PyPy

In this section we evaluate Hypothesis 1 – in essence, does optimizing across the boundary between PyPy and SQPyte lead to a significant performance increase? – and Hypothesis 3 – in essence, does exposing type information in the `flags` attribute increase performance? As well as benchmarking SQLite, SQPyte, and H2 (as in Section 5), we also benchmark two SQPyte variants: $\text{SQPyte}_{\text{no-inline}}$ turns off inlining between SQPyte and PyPy and $\text{SQPyte}_{\text{no-flags}}$ turns off the type flags optimisations of Section 3.3.

7.1 Micro-benchmarks for PyPy Integration

The TPC-H benchmarks measure SQL performance in isolation, but tell us nothing about the performance of a PL calling a DBMS. Indeed, to the best of our knowledge, there are no relevant benchmarks in this style. In order to test Hypothesis 1, we were therefore forced to create 6 micro-benchmarks, each designed to pass large quantities of data across the PL / DBMS boundary. While they are not necessarily completely realistic programs, they exemplify common idioms in larger programs (see Figures 3 and 13). The micro-benchmarks are as follows:

Table 2 How often the micro-benchmarks cross the boundary between Python and the database, and how many values are converted across the boundary in total. In most cases, the 'values converted' is a whole-number multiple of 'crossings'. `pyfunction` crosses the PL / DBMS boundary twice per iteration, with one crossing returning one value, the other two. `pythonjoin` has a similar, though more complex, pattern of crossings to `pyfunction`.

Benchmark	Crossings	Values converted
select	6 001 217	18 003 645
innerjoin	800 002	1 600 000
pythonjoin	4 000 002	4 800 000
pyfunction	12 002 434	18 003 645
pyaggregate	6 001 218	12 002 431
filltable	200 004	400 000

select is the running example of Figure 3. The DBMS query iterates over three columns of a table, returning them to Python, which performs arithmetic operations on the results.

innerjoin joins 3 tables with an inner join and returns the resulting tuples to Python, which are then stored into a hashmap.

pythonjoin implements a semantically equivalent join to the *innerjoin* benchmark, but does so in Python rather than using the DBMS. The Python code iterates over 1 of the tables, on each iteration executing 2 sub-queries for the other 2 tables. On the Python side the tuples are stored into a hashmap. The core part of this micro-benchmark can be seen in Figure 13.

pyfunction models calling back to a Python function from SQL. An `abs` function is defined in pure Python. The SQL query then iterates over all rows in a table, calling `abs` on one column, and returning that column's value to Python, which then sums all the elements.

pyaggregate models calling an aggregate defined in Python. A `sum` aggregate is defined in pure Python (as in Figure 12) and used to sum one column of a table.

filltable first adds 100,000 rows to a two-column table, with each row being added in a single SQL query. A single SQL query then reads all of the added rows back out again.

Each benchmark has a Java equivalent such that we can run it with H2. All micro-benchmarks use the TPC-H dataset from Section 5, with the exception of the `filltable` micro-benchmark which creates, writes, and reads from its own tables. Table 2 shows how often each micro-benchmark crosses between DBMS and PL, and how many values are converted between the DBMS and PL.

7.2 Results and Evaluation

The results of the micro-benchmarks are shown in Figure 3. They show that on these conversion-heavy queries SQPyte outperforms SQLite by a factor of $3.367 \pm 0.0637\times$ on average. Table 2 shows how often each micro-benchmark crosses the DBMS / PL boundary. As predicted by Hypothesis 1, the more often a micro-benchmark crosses the boundary (as shown in Table 2), the greater SQPyte's advantage.

H2, in contrast, is significantly slower on these benchmarks – $30.285 \pm 0.3515\times$ – than on the TPC-H benchmarks. We believe that this is because HotSpot is unable to optimize effectively across the PL / DBMS boundary. Unfortunately, definitively verifying that this is the cause is impossible, as we cannot selectively turn on and off the relevant HotSpot optimisations. However, the magnitude of the effect strongly suggests that simply having

Table 3 Results of the micro-benchmark set. The table shows absolute times in seconds, as well as the relative factor of each VM normalized to SQPyte. The last row contains the geometric mean of the normalized factors. Micro-benchmarks where SQLite or H2 are faster than SQPyte are shown in bold. Micro-benchmarks where SQLite or H2 are, within the confidence interval, equivalent in performance to SQPyte are shown in grey.

Benchmark	SQPyte	SQLite	H2
select (s)	0.772 ± 0.0081	3.382 ± 0.0114	73.095 ± 0.9489
×		4.382 ± 0.0515	94.662 ± 1.6645
innerjoin (s)	0.578 ± 0.0030	0.913 ± 0.0032	20.957 ± 0.1636
×		1.579 ± 0.0102	36.268 ± 0.3472
pythonjoin (s)	1.397 ± 0.0061	3.332 ± 0.0862	9.292 ± 0.0776
×		2.385 ± 0.0585	6.651 ± 0.0628
pyfunction (s)	0.580 ± 0.0027	3.861 ± 0.0930	55.298 ± 0.3916
×		6.661 ± 0.1678	95.402 ± 0.8443
pyaggregate (s)	0.542 ± 0.0289	2.558 ± 0.0712	8.218 ± 0.0387
×		4.730 ± 0.2893	15.167 ± 0.7735
filltable (s)	0.067 ± 0.0013	0.188 ± 0.0149	1.565 ± 0.0443
×		2.805 ± 0.2336	23.342 ± 0.8382
Geometric mean ×		3.367 ± 0.0648	30.283 ± 0.3563

both PL and DBMS running on the same VM is not sufficient to optimize across the PL / DBMS boundary effectively.

In order to understand the cause of SQPyte's good performance on the micro-benchmarks, we created $\text{SQPyte}_{\text{no-inline}}$, a simple variant of SQPyte which disables all inlining between SQPyte and PyPy. Note that although no inlining occurs, traces in $\text{SQPyte}_{\text{no-inline}}$ are still created on both sides of the PL / DBMS boundary, so we are able to make a sensible comparison between SQPyte and $\text{SQPyte}_{\text{no-inline}}$.

The resulting figures are shown in the second columns of Tables 4 and 5. As expected, there is no statistical difference in the performance of SQPyte and $\text{SQPyte}_{\text{no-inline}}$ on the TPC-H benchmarks ($0.3 \pm 1.99\%$)—the only Python code in these benchmarks is that used to consume the results of a query, ensuring that the database definitely produces the results.

The micro-benchmarks are rather different, with $\text{SQPyte}_{\text{no-inline}}$ being $2.388 \pm 0.0350\times$ slower than SQPyte. This shows that inlining is the single biggest part of the speed benefit of SQPyte relative to SQLite. The only micro-benchmark that is relatively little affected is `innerjoin`, which is $1.266 \pm 0.0079\times$ slower than SQPyte. This is because most of the work in the benchmark is involved in the table joins, which happen entirely in the DBMS. In contrast, `pyfunction`, which crosses the boundary twice per iteration (once from Python to the DBMS, and then from the query calling back to Python) sees a large slowdown in $\text{SQPyte}_{\text{no-inline}}$ of $3.501 \pm 0.0555\times$.

In summary, not only does SQPyte give a significant performance increase when the DBMS / PL boundary is crossed regularly, but we can see that inlining is the major factor in this. This strongly validates Hypothesis 1.

Table 4 Results of the micro-benchmark set. $\text{SQPyte}_{\text{no-inline}}$ disables inlining across the database-programming language boundary, $\text{SQPyte}_{\text{no-flags}}$ disables the optimisation that reasons about the `flags` attribute of the `Mem` structures. The table shows average absolute times in seconds, as well as the factor of SQPyte normalized to each of the other VMs. The last row contains the geometric mean of the normalized factors. Micro-benchmarks where SQLite or H2 are faster than SQPyte are shown in bold. Micro-benchmarks where SQLite or H2 are, within the confidence interval, equivalent in performance to SQPyte are shown in grey.

Benchmark	SQPyte	$\text{SQPyte}_{\text{no-inline}}$	$\text{SQPyte}_{\text{no-flags}}$
select (s)	0.772 ± 0.0081	1.948 ± 0.0190	0.795 ± 0.0029
×		2.524 ± 0.0383	1.030 ± 0.0119
innerjoin (s)	0.578 ± 0.0030	0.732 ± 0.0025	0.579 ± 0.0017
×		1.266 ± 0.0079	1.003 ± 0.0060
pythonjoin (s)	1.397 ± 0.0061	2.961 ± 0.0796	1.423 ± 0.0117
×		2.117 ± 0.0605	1.019 ± 0.0099
pyfunction (s)	0.580 ± 0.0027	2.029 ± 0.0302	0.605 ± 0.0021
×		3.501 ± 0.0551	1.044 ± 0.0062
pyaggregate (s)	0.542 ± 0.0289	1.380 ± 0.0619	0.529 ± 0.0023
×		2.547 ± 0.1862	0.976 ± 0.0498
filltable (s)	0.067 ± 0.0013	0.206 ± 0.0017	0.069 ± 0.0016
×		3.072 ± 0.0670	1.032 ± 0.0327
Geometric mean ×		2.388 ± 0.0346	1.017 ± 0.0113

8 Evaluation of Hypothesis 3: The Effect of Optimizing the `flags` Attribute

In order to see how much the optimisation of the `flags` attribute of the `Mem` struct described in Section 3.3 helps, we created a version $\text{SQPyte}_{\text{no-flags}}$ of SQPyte that disables this optimisation completely and reran all benchmarks. The results are shown in the last columns of Tables 4 and 5.

We expected that turning off the `flags` optimisations would slow execution down, and that it would account for much of the performance benefit not accounted for by inlining in Section 7.2. On the TPC-H benchmarks, there is no statistically observable effect (a slowdown $1.0 \pm 2.41\%$). On the micro-benchmarks, the slowdown is statistically significant ($1.7 \pm 1.112\%$), but only very marginally.

These results were not what we expected, and lead us to reject Hypothesis 3.

9 Threats to Validity

Benchmarks can only provide a partial view of a system's overall performance, and thus don't necessarily reflect the behaviour of more realistic settings and workloads. The TPC-H benchmarks are widely used, though the micro-benchmarks are our own creations, and we may unintentionally have created micro-benchmarks which unduly flatter SQPyte.

When porting SQLite's interpreter to SQPyte, we only ported those parts enabled in the default build of SQLite. Since some parts are tangled up in C `#ifdef`s, we may have unintentionally misclassified one or more of these parts. Appendix A contains a complete list of the parts we did not port, so that readers can verify our choices.

There is a subtle difference between PyPy calling SQLite and SQPyte: in the former case, PyPy uses a C FFI (the `cffi` module in PyPy) to interface with SQLite; in the latter, PyPy

Table 5 Further variants of SQPyte running TPC-H. For a description of the columns see Table 4.

Benchmark	SQPyte	$\text{SQPyte}_{\text{no-inline}}$	$\text{SQPyte}_{\text{no-flags}}$
Query 1 (s)	6.929 ± 0.0352	6.903 ± 0.0137	7.082 ± 0.0239
×		0.996 ± 0.0055	1.022 ± 0.0064
Query 2 (s)	0.298 ± 0.0098	0.296 ± 0.0065	**0.283 ± 0.0039**
×		0.995 ± 0.0385	**0.953 ± 0.0345**
Query 3 (s)	2.933 ± 0.0329	2.922 ± 0.0136	2.957 ± 0.0379
×		0.996 ± 0.0124	1.008 ± 0.0184
Query 4 (s)	0.345 ± 0.0038	0.346 ± 0.0038	0.346 ± 0.0041
×		1.002 ± 0.0159	1.001 ± 0.0169
Query 5 (s)	1.111 ± 0.0145	1.114 ± 0.0117	1.097 ± 0.0176
×		1.003 ± 0.0185	0.987 ± 0.0221
Query 6 (s)	0.701 ± 0.0081	0.693 ± 0.0016	0.702 ± 0.0014
×		0.990 ± 0.0118	1.001 ± 0.0117
Query 7 (s)	2.630 ± 0.0070	2.637 ± 0.0178	2.657 ± 0.0105
×		1.003 ± 0.0075	1.010 ± 0.0049
Query 8 (s)	2.510 ± 0.0141	2.494 ± 0.0167	2.499 ± 0.0270
×		0.994 ± 0.0088	0.996 ± 0.0124
Query 9 (s)	10.062 ± 0.0448	10.102 ± 0.0227	10.138 ± 0.0766
×		1.004 ± 0.0051	1.007 ± 0.0090
Query 10 (s)	0.019 ± 0.0056	0.020 ± 0.0057	0.023 ± 0.0089
×		1.079 ± 0.4796	1.237 ± 0.6278
Query 11 (s)	0.604 ± 0.0071	0.602 ± 0.0093	0.600 ± 0.0073
×		0.998 ± 0.0205	0.994 ± 0.0178
Query 12 (s)	0.938 ± 0.0062	0.937 ± 0.0067	0.934 ± 0.0045
×		0.999 ± 0.0099	0.995 ± 0.0083
Query 13 (s)	2.721 ± 0.0135	2.767 ± 0.0420	2.767 ± 0.0461
×		1.017 ± 0.0164	1.017 ± 0.0178
Query 14 (s)	0.792 ± 0.0102	0.785 ± 0.0048	0.793 ± 0.0101
×		0.991 ± 0.0151	1.001 ± 0.0196
Query 15 (s)	20.636 ± 0.2254	20.437 ± 0.1008	20.674 ± 0.3404
×		0.991 ± 0.0120	1.002 ± 0.0210
Query 16 (s)	0.410 ± 0.0074	0.416 ± 0.0088	0.443 ± 0.0380
×		1.014 ± 0.0295	1.079 ± 0.0972
Query 17 (s)	0.107 ± 0.0008	0.107 ± 0.0007	0.108 ± 0.0008
×		1.001 ± 0.0102	1.009 ± 0.0112
Query 18 (s)	2.449 ± 0.0144	2.441 ± 0.0043	2.485 ± 0.0145
×		0.997 ± 0.0062	1.015 ± 0.0091
Query 19 (s)	8.140 ± 0.1333	8.082 ± 0.0744	**7.883 ± 0.0611**
×		0.993 ± 0.0191	**0.968 ± 0.0182**
Query 20 (s)	80.386 ± 0.2692	80.801 ± 0.3028	80.542 ± 0.3804
×		1.005 ± 0.0054	1.002 ± 0.0061
Query 21 (s)	8.661 ± 0.0347	8.684 ± 0.0782	**8.572 ± 0.0340**
×		1.003 ± 0.0101	**0.990 ± 0.0059**
Query 22 (s)	0.087 ± 0.0036	0.087 ± 0.0037	0.084 ± 0.0020
×		0.998 ± 0.0601	0.959 ± 0.0465
Geometric mean ×		1.003 ± 0.0197	1.010 ± 0.0237

simply imports the SQPyte system as an RPython module. There is thus the potential of additional overhead when PyPy calls SQLite compared to when it calls SQPyte. We examined the PyPy traces for the case when it calls SQLite, and verified that the overhead is extremely small (a small handful of machine code instructions), and insignificant relative to the difference to the two systems.

10 Related Work

We split our discussion of related work into two sections: optimizing SQL with code generation; and optimizing the interactions between PLs and DBMSs.

10.1 Optimizing Execution SQL with Code Generation

Many databases use the *iterator model* for query execution [20, 11] which, in essence, is equivalent to an AST interpreter in PL implementation. There have been many attempts to generate code from query plans to reduce the overheads of the iterator model. This started with very early databases such as System R [6]. Most of these approaches require code generators to be written by hand. In contrast, SQPyte's meta-tracing JIT compiler implicitly implements the semantics of the system by tracing the RPython interpreter.

Rao et al. [24] describe a relational, Java-based, in-memory database that, for each query, dynamically generates new query-specific code. They created two versions of the query planner: an interpreted one using the iterator model and a compiled one. They demonstrated that using the compiled version removed the overhead of virtual functions in the interpreted version. In addition, the Java JIT compiler was much better at optimizing the generated code for each query than the interpreted version. On average, the compiled queries in their benchmark ran twice as fast as the interpreter.

Krikellas et al. [17] generate C code from queries and load the compiled shared libraries to execute them. Their compilation process dynamically instantiates carefully handwritten templates to create source code specific for a given query and hardware. The performance of their dynamically generated evaluation plans are comparable to hard-coded equivalents.

Neumann [23] describes an approach where the query is compiled to machine code using the LLVM compiler [18]. When generating code, the approach attempts to keep data in registers for as long as possible. Similar to how SQPyte interacts with the existing SQLite C code, this system also preserves complex parts of the database in C++ and calls into the C++ code from the LLVM code as needed. The resulting system is $2-4\times$ faster than the other databases benchmarked.

Klonatos et al. [15, 16, 25] use generative programming techniques in Scala to dynamically compile queries using a Scala SQL query engine into C. This technique implicitly inlines parts of the DBMS into the query and shows good performance on the TPC-H benchmark suite relative to the DBX DBMS. However, because code written in Scala cannot inline the generated C, there is no equivalent of the PyPy / SQPyte bridge we implemented.

Haensch et al. [14] describe an automatic operator model specialization system. Their system uses a general LLVM-based specialization component to dynamically optimize the execution of query plans in the operator model with the help of partial evaluation. Certain query operator fields are marked as immutable, allowing the specializer to aggressively inline and optimize the query plan execution. This approach suffers somewhat from having to perform its optimisations at the (rather low-level) LLVM IR, at which point a lot of useful high-level information has been lost. Both this approach and SQPyte use interpreters as

the basis of the run-time code generation, though Haensch et al.'s interpreters are closer in style to the AST-based partial evaluation system of Truffle [29].

10.2 Optimizing Language-Database Interaction

Grust et al. [13] created the Ferry glue language which serves as an intermediate language to translate subsets of other languages to SQL. That is various front-end languages can translate to that intermediate language, which is then lowered into SQL code. The goal is to reduce the impedance mismatch between languages and database when programming and to improve the efficiency of the interaction. Ferry influenced several works in other languages: Garcia et al. [9] developed a Scala plugin that enables programmers to translate Scala-level constructs to Ferry; Grust et al. [12] introduced Switch which uses Ferry-like translation principles to allow seamless integration of Ruby and Ruby on Rails with the DBMS; Giorgidze et al. [10] designed and implemented a Haskell library for database-supported program execution; and Schreiber et al. [26] created a Ferry-based LINQ-to-SQL provider. Since one of the main goals of Ferry is to reduce the number of times the PL / DBMS boundary is crossed, the approach is complementary to the SQPyte approach of reducing the cost of the boundary crossings.

Mattis et al. [21] describe columnar objects, which is an implementation of an in-memory column store embedded into Python together with a seamless integration into the Python object system. With the help of the PyPy JIT compiler they produce efficient machine code for Python code that queries the data store. Compared to SQPyte their approach offers a much deeper integration of the database implementation into the host language, at the cost of having to implement the data store from scratch.

Unipycation by Barrett et al. [2] is a language composition of Prolog and Python that uses meta-tracing to reduce the overhead of crossing the boundary between the two languages. It composes together PyPy with Pyrolog, a Prolog interpreter written in RPython. As with SQPyte, the most effective optimisation is inlining.

11 Conclusion

This paper's major result is that there are substantial, and previously missed, opportunities for optimizing across the PL / DBMS boundary. We achieved a significant performance increase by inlining queries from SQL into PyPy. Furthermore, most of this performance increase came from tracing's natural tendency to inline—our attempts to add more complex dynamic typing optimisations had little effect.

Those who wish to apply our approach to other embedded DBMSs can take heart from this: a relatively simple conversion of parts of an interpreter implemented in C into a meta-tracing language is highly effective. We estimate that we spent at least 8 person-months on the SQPyte implementation, with perhaps half of that spent on the `flags` optimisation, and a further 2 person-months on the PyPy / SQPyte bridge. For this relatively moderate effort – certainly compared to the much greater work put into both SQLite and PyPy – we were able to substantially improve performance for queries that regularly cross the PL / DBMS boundary. While we were only able to marginally increase the performance of stand-alone SQL queries, we did not encounter any examples where SQPyte is slower than SQLite. This suggests that SQPyte, or a similar system based on SQLite, may be useful to a wider range of users.

Our approach of incrementally replacing SQLite's C code with RPython had an interesting trade-off. It made initial development easier, since we always had a running system.

However, it had disadvantages which became more apparent in later stages of development. Most obviously, since it necessitated keeping core data-structures in C, we hobbled the trace optimizer somewhat. The best – or, from our perspective, worst – example of this is the `flags` optimisation which, despite significant effort, ended up slightly slowing our system down. We suspect that porting more of these data-structures, and the code that relies on them, into RPython would enable further performance increases. Indeed, were we to tackle SQPyte from scratch, we might place less emphasis on keeping interpreter data-structures in C—we conjecture that in several places we might have incurred less effort on our part if we had ported more C data-structures into RPython.

A secondary, and largely implicit result, is that we have shown that it is possible to take an existing interpreter in C and replace relevant parts of it with RPython, creating a meta-tracing VM. To the best of our knowledge, the first time this has been done. It may be possible to apply this technique to other systems (including non-DBMSs), with minor adjustments.

Acknowledgements. We thank Geoff French for comments.

References

1 Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI*, 2000.

2 Edd Barrett, Carl Friedrich Bolz, and Laurence Tratt. Approaches to interpreter composition. *Comput. Lang. Syst. Str.*, abs/1409.0757, 2015. URL: `http://arxiv.org/abs/1409.0757`.

3 Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: A trace-based JIT compiler for CIL. In *OOPSLA*, 2010.

4 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *ICOOOLPS*, 2009.

5 Carl Friedrich Bolz and Laurence Tratt. The impact of meta-tracing on VM design and implementation. *To appear J. SCICO*, 2014.

6 Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of System R. *Commun. ACM*, 24(10), 1981.

7 Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *OOPSLA*, 1989. URL: `http://portal.acm.org/citation.cfm?id=74884`.

8 Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *VEE*, 2006.

9 Miguel Garcia, Anastasia Izmaylova, and Sibylle Schupp. Extending Scala with database query capability. *JOT*, 9(4), 2010. URL: `http://www.jot.fm/contents/issue_2010_07/article3.html`.

10 George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. Haskell boards the Ferry: Database-supported program execution for Haskell. In *IFL*, 2010. URL: `http://dl.acm.org/citation.cfm?id=2050135.2050136`.

11 Goetz Graefe and William J. McKenna. The Volcano optimizer generator: extensibility and efficient search. In *ICDE*, 1993.

12 Torsten Grust and Manuel Mayr. A deep embedding of queries into Ruby. In *ICDE*, 2012.

13 Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: Database-supported program execution. In *SIGMOD*, 2009.
14 Carl-Philip Haensch, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. Plan operator specialization using reflective compiler techniques. In *BTW*, 2015.
15 Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10), 2014. URL: `http://www.vldb.org/pvldb/vol7/p853-klonatos.pdf`.
16 Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Errata for "Building efficient query engines in a high-level language": PVLDB 7(10):853-864. *PVLDB*, 7(13), 2014.
17 Konstantinos Krikellas, Stratis D. Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.
18 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.
19 Anatole Le, Ondřej Lhoták, and Laurie Hendren. Using inter-procedural side-effect information in JIT optimizations. In *CC*, 2005.
20 Raymond A Lorie. XRM - an extended (n-ary) relational memory. Technical Report G320-2096, IBM Research Report, 1974.
21 Toni Mattis, Johannes Henning, Patrick Rein, Robert Hirschfeld, and Malte Appeltauer. Columnar objects: Improving the performance of analytical applications. In *Onward!*, 2015.
22 James George Mitchell. *The design and construction of flexible and efficient interactive programming systems*. PhD thesis, Carnegie Mellon University, 1970.
23 Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9), 2011.
24 Jun Rao, Hamid Pirahesh, C. Mohan, and Guy Lohman. Compiled query execution engine using JVM. In *ICDE*, 2006.
25 Tiark Rompf and Nada Amin. Functional pearl: A SQL to C compiler in 500 lines of code. In *ICFP*, 2015.
26 Tom Schreiber, Simone Bonetti, Torsten Grust, Manuel Mayr, and Jan Rittinger. Thirteen new players in the team: A Ferry-based LINQ to SQL provider. *PVLDB*, 3(1-2), 2010.
27 Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic native optimization of interpreters. In *IVME*, 2003.
28 Transaction Processing Performance Council. TPC-H, a decision support benchmark. http://www.tpc.org/tpch, 2015.
29 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Onward!*, 2013.
30 Alexander Yermolovich, Christian Wimmer, and Michael Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. In *DLS*, 2009.

A Unported Aspects of SQLite

When porting SQLite C code to RPython, we did not port the following aspects:

- Assert statements, which are removed by the C compiler.
- Statements related to tests and debugging, which expand to nothing in production builds:
  - `VdbeBranchTaken`
  - `REGISTER_TRACE`
  - `SQLITE_DEBUG`
  - `memAboutToChange`

  - `UPDATE_MAX_BLOBSIZE`
- Blocks of `#ifdef` and `#ifndef`, which are usually not included in default production builds:
  - `SQLITE_DEBUG`
  - `SQLITE_OMIT_FLOATING_POINT`
- We assumed `SQLITE_THREADSAFE` to be false, which SQLite recommends for best single-threaded performance.
- We decided to compile and port with `SQLITE_OMIT_PROGRESS_CALLBACK` turned on. Usually SQLite makes it possible to register a progress callback that is called every n opcodes. We plan to implement this in the future. Note that in our evaluation, we compared SQPyte to SQLite with callbacks similarly omitted, thus ensuring an apples-to-apples comparison.

Reference Capabilities for Concurrency Control*

Elias Castegren[1] and Tobias Wrigstad[2]

1 **Uppsala University, Sweden,** `Elias.Castegren@it.uu.se`
2 **Uppsala University, Sweden,** `Tobias.Wrigstad@it.uu.se`

Abstract

The proliferation of shared mutable state in object-oriented programming complicates software development as two seemingly unrelated operations may interact via an alias and produce unexpected results. In concurrent programming this manifests itself as data-races. Concurrent object-oriented programming further suffers from the fact that code that warrants synchronisation cannot easily be distinguished from code that does not. The burden is placed solely on the programmer to reason about alias freedom, sharing across threads and side-effects to deduce where and when to apply concurrency control, without inadvertently blocking parallelism.

This paper presents a reference capability approach to concurrent and parallel object-oriented programming where all uses of aliases are guaranteed to be data-race free. The static type of an alias describes its possible sharing without using explicit ownership or effect annotations. Type information can express non-interfering deterministic parallelism without dynamic concurrency control, thread-locality, lock-based schemes, and guarded-by relations giving multi-object atomicity to nested data structures. Unification of capabilities and traits allows trait-based reuse across multiple concurrency scenarios with minimal code duplication. The resulting system brings together features from a wide range of prior work in a unified way.

1998 ACM Subject Classification D.3.3 [Programming Languages] Language Constructs and Features – Classes and Objects

Keywords and phrases Type systems, Capabilities, Traits, Concurrency, Object-Oriented

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.5

1 Introduction

Shared mutable state is ubiquitous in object-oriented programming. Sharing can be more efficient than copying, especially when large data structures are involved, but with great power comes great responsibility: unless sharing is carefully maintained, changes through a reference might propagate unexpectedly, objects may be observed in an inconsistent state, and conflicting constraints on shared data may inadvertently invalidate invariants, etc. [28].

Multicore programming stresses proper control of sharing to avoid interference or data-races[1] and to synchronise operations on objects so that their changes appear atomic to the system. Concurrency control is a delicate balance: locking too little opens up for the aforementioned problems. Locking too much loses parallelism and decreases performance.

For example, parallelism often involves using multiple threads to run many tasks simultaneously without any concurrency control. This requires establishing non-interference by considering all the objects accessed by the tasks at any level of indirection.

* This work was partially funded by the Swedish Research Council project Structured Aliasing, the EU project FP7-612985 Upscale (`http://www.upscale-project.eu`), and the Uppsala Programming Multicore Architectures Research Centre (UPMARC)

[1] Two concurrent operations accessing the same location (read–write or write–write) without any synchronisation is a data-race. Non-interference allows only read–read races and no locks.

Mainstream programming languages place the burden of maintaining non-interference, acquiring and releasing locks, reasoning about sharing, etc. completely on the (expert) programmer. This is unreasonable, especially considering the increasing amount of parallelism and concurrency in applications in the age of multicore and manycore machines [6].

In this paper, we explore a reference capability approach to sharing objects across threads. A capability [31, 33] is a token that grants access to a particular resource, in our case objects. Capabilities present an alternative approach to tracking and propagating computational effects to check interference: capabilities assume exclusive access to their governed resources, or only permit reading. Thus, holding a capability implies the ability to use it fully without fear of data-races. This shifts reasoning from use-site of a reference to its creation-site.

We propose a language design that integrates capabilities with traits [39], *i.e.,* reusable units from which classes are constructed. This allows static checking at a higher level of abstraction than *e.g.,* annotations on individual methods. A *mode* annotation on the trait controls how exclusivity is guaranteed, *e.g.,* by completely static means such as controlling how an object may be referenced, or dynamically, by automatically wrapping operations in locks. A trait can be combined with different modes to form different capabilities according to the desired semantics: thread-local objects, immutable objects, unsharable linear objects, sharable objects with built-in concurrency control, or sharable objects for which locks must be acquired explicitly. This extends the reusability of traits across concurrency scenarios.

The sharing or non-sharing of a value is visible statically through its type. Types are formed by composing capabilities. Composition operators control how the capabilities of a type may share data, which ultimately controls whether an object can be aliased in ways that allow manipulation in parallel. Hiding a type's capabilities allows changing its aliasing restrictions. For example, hiding all mutating capabilities creates a temporarily immutable object which is consequently safe to share across threads (*cf.,* [9]).

Ultimately, with a small set of primitives—differently moded capabilities and composition operators—working in concert, the resulting system brings together many features from prior work: linear types [41, 23] and unique references [27, 34, 8, 17], regions [25], ownership types [16], universe types [22] and (fractional) permissions [9, 42]. As far as the authors are aware, there is no other *single* system that can express all of these concepts in a unified way.

This paper makes several contributions to the area of type-driven concurrency control:

- We present a framework for defining capabilities which work in concert to express a wide variety of concepts from prior work on alias control. The novel integration of capabilities with traits extends trait-based reuse across different concurrency scenarios without code duplication. Traits are guaranteed to be data-race free or free from any interference, which simplifies their implementation and localises reasoning. A single keyword controls this aspect. We support both internal and external locking schemes for data (Section 3–4).
- We formalise our system in the context of the language κ (pronounced kappa), state the key invariants of our system (safe aliasing, data-race freedom, strong encapsulation, thread-affinity and partial determinism) and prove them sound (Section 6–7).

The full proofs, dynamic semantics and a few longer code examples can be found in the accompanying technical report [15].

2 Problem Overview

Object-oriented programs construct graphs of objects whose entangled structure can make seemingly simple operations hard to reason about. For example, the behaviour of the following program (adapted from [28]) manipulating two counters `c1` and `c2` depends on

whether `c1` and `c2` may alias, which may only be true for some runs of the program.

```
assert c1.value() == 42; c1.inc(); c2.inc(); assert c1.value() == 43;
```

If `c1` and `c2` *always alias*, we may reason about the sequential case, but if `c2.inc()` is performed by *another thread*, the behaviour is affected by the scheduling of `c2.inc()`, and whether `inc()` itself is thread-safe. While aliasing is possible without sharing across threads, sharing across threads is not possible without aliasing. With this in mind, we move on to three case studies to discuss some of the challenges facing concurrent object-oriented programming.

2.1 Case Study: Simple Counters

To achieve thread-safety for a counter implemented in Java we can make the `inc()` method synchronised to ensure only one thread at a time can execute it. While this might seem straightforward, there are at least three problems with this approach:

1. Additional lock and unlock instructions for each increment will be inserted regardless of whether they are necessary or not – synchronising an unaliased object is a waste.
2. Making the object thread-safe does not help protect an instance from being shared, which might have correctness implications (*e.g.,* non-determinism due to concurrent accesses).
3. Unless the `value()` method is also synchronised, concurrent calls to `inc()` and `value()` may lead to a data-race, which might lead to a perception of lost increments.

In 1. and 2., the underlying problem is distinguishing objects shared across threads from thread-local objects as only the former needs synchronisation. Using two different classes for shared and unshared counters are possible, but leads to code duplication. Furthermore, if a counter is shared indirectly, *i.e.,* there is only one counter but its containing object is shared, the necessary concurrency control might be in the container. Establishing and maintaining such a "guarded-by property" warrants tool support.

In 3., the underlying problem is the absence of machinery for statically checking that all accesses to data are sufficiently protected. This might not be easy, for example, excluding data-races in methods inherited from a super class that encapsulates its locking behaviour.

2.2 Case Study: Data Parallelism and Task Parallelism

The counter exemplifies concurrent programming which deals with asynchronous behaviour and orchestration of operations on shared objects. In contrast, parallelism is about optimisation with the goal of improving some aspect of performance.

Consider performing the operations f_1 and f_2 on all elements in a collection E. A data parallel approach might apply $f_1(f_2(e))$ in parallel to all $e \in E$. In contrast, a task parallel approach might execute $f_1(e_1); \ldots; f_1(e_n)$ and $f_2(e_1); \ldots; f_2(e_n)$ as two parallel tasks.

Both forms of parallelism requires proper alias management to determine whether $f_1(e_i)$ and $f_2(e_j)$ may safely execute in parallel. When $i = j$, we must determine what parts of an object's interface might be used concurrently. When $i \neq j$, we must reason about the possible overlapping states of (the different) elements e_i and e_j. Furthermore, unless $f_1(e)$ (or $f_1(f_2(e))$) is safe to execute in parallel on the same object, we must exclude the possibility that E contains duplicate references to the same object.

If f_1 and f_2 only perform reads, any combination is trivially safe. However, correctly categorising methods as accessors or mutators manually can be tricky, especially if mutation happens deep down inside a nested object structure, and a method which may logically

only read might perform mutating operations under the hood for optimisation, telemetry, etc. Extending the categorisation of methods to include mutation of disjoint parts further complicates this task. Further, as software evolves, a method's categorisation might need to be revisited, even as a result of a non-local change (*e.g.,* in a superclass).

2.3 Case Study: Vector vs. ArrayList in Java

As a final case study, consider the `ArrayList` and `Vector` classes from the Java API. While both implement a list with comparable interfaces, vectors are thread-safe whereas array lists are not. There are several consequences of this design:

1. Vector objects lock individual operations. This requires multiple acquires and releases for compound operations (*e.g.,* when using an external iterator to access multiple elements).
2. The reliance on Java objects' built-in synchronisation excludes concurrent reads.
3. Just like the counter above, even thread-local vectors pay the price of synchronisation.

As a result, `ArrayList` is commonly favoured over `Vector` despite the fact that this requires locks to be acquired correctly for each use, rather than once if built into the data structure.

A lock that allows multiple concurrent reads (a readers–writer lock) would allow both vectors and array lists to be used efficiently and safely in parallel. This distinction adds an extra dimension of locking and requires categorising methods as accessors/mutators.

Summary. The examples above illustrate a number of challenges facing programmers doing concurrent and parallel programming in object-oriented languages. In summary:

- Code that needs synchronisation for data-race freedom is indistinguishable from code that does not. The same holds for code correctly achieving non-interference.
- Conservatively adding locks to all data structure definitions or all uses of a data structure hurts performance.
- Using locks to exclude conflicting concurrent accesses is non-trivial and requires reasoning about aliasing and program-wide sharing of data structures. The same reasoning is required for partitioning a data structure across multiple threads for parallel operations on disjoint parts, or specifying read-only operations.
- The need for concurrency control varies across different usage scenarios. Building concurrency control into data structures generates overhead or leads to code duplication (one thread-safe version and one which is not). Leaving concurrency control in the hands of clients instead opens up for under-synchronisation and concurrency bugs.
- The need for alias control varies across different usage scenarios. At times, thread-locality or even stronger aliasing restrictions are desirable, for example to avoid locks or non-determinism, or to unlock compiler optimisations or simplify verification. At other times, sharing is required. The sharing requirements of a single object could even vary over time.

We now describe our reference capability system which addresses all of these problems.

3 Capabilities for Concurrency Control

Our starting point for this work is to unify references and capabilities. A capability is a handle to a resource—a part of or an entire object or aggregate (an object containing other objects). A capability exposes a set of operations, which can be used to access its resource

without possibility of data-races. Granting and revoking capabilities corresponds to creating and destroying aliases. Capabilities' *modes* controls how they may be shared across threads:

Exclusive capabilities denote resources that are exclusive to one thread so that accesses are trivially free from any interference from other threads. There are two exclusive modes: `linear`, used for resources to which there is only a single handle in the program, and `thread`, which allows sharing, but only within one single thread. `linear` capabilities must be fully transferred from one thread in order to be used by another thread.

Safe capabilities denote resources that can be arbitrarily shared (*e.g.,* across multiple threads). There are two safe modes: `locked`, causing operations to be implicitly guarded by locks, and `read` which do not allow causing or directly observing mutation. Safe capabilities guarantee data-race freedom.

Subordinate capabilities (the mode `subordinate`) denote resources that are encapsulated inside some object and therefore inherit its protection against data-races or interference. Subordinate capabilities are similar to `rep` or `owner` in ownership types [16].

Unsafe capabilities (the mode `unsafe`) denote arbitrarily shared resources which are unsafe to use concurrently without some means of concurrency control. Accesses to unsafe capabilities must be wrapped in explicit locking instructions.

Linear capabilities impose transfer semantics on assignment. We adopt destructive reads [27] here for simplicity. This means that reading a variable holding a linear capability has the side-effect of updating it with `null`. Methods in `locked` capabilities automatically get acquire and release instructions, providing *per-method* atomicity. For `unsafe` capabilities locking must be done manually, providing *scoped* atomicity (the duration of the lock). Although straightforward, for simplicity we do not allow manual locking of `locked` in this presentation.

Types are compositions of one or more capabilities (*cf.,* Section 3.3) and expose the union of their operations. The modes of the capabilities in a type control how resources of that type can be aliased. The compositional aspect of our capabilities is an important difference from normal type qualifiers (*cf., e.g.,* [24]), as accessing different parts of an object through different capabilities in the same type gives different properties.

Exclusive and read capabilities guarantee *non-interference* and enable deterministic parallelism. Safe capabilities guarantee the absence of *data-races*, *i.e.,* concurrent write–write or read–write operations to the same memory location, but do not exclude *race-conditions*, *e.g.,* two threads competing for the same lock. This means that programs will be thread-safe, only one thread can hold the lock, but not necessarily deterministic—the order in which competing threads acquire a lock is controlled by factors external to the program. This also means that capabilities using locks do not exclude the possibility of deadlocks.

3.1 Capability = Trait + Mode

We present our capabilities system through κ, a Java-like language that uses traits [39] in place of inheritance for object-oriented reuse. A κ capability corresponds to a trait with some required fields, provided methods, and a mode. For the reader not familiar with traits, a trait can be thought of as an abstract class whose *fields* are abstract and must be provided by a concrete subclass—see Figure 4 for a code example of traits and classes. An important property of κ is that an *implementer of a trait can assume freedom from data-races or interference*, which enables sequential reasoning for all data that the trait *owns*, (its subordinate capabilities), plus reachable exclusive capabilities. A trait's mode controls how data-race freedom or non-interference is guaranteed. For example, prohibiting aliases to cross thread boundaries or inserting locks at compile-time in its methods.

The mode of a trait is either *manifest* or must be given wherever the trait is included by a class. A manifest mode is part of the declaration of the trait, meaning the trait defines a single capability. As an example of this, consider the capability **read** `Comparable` which provides compare methods to a class which do not mutate the underlying object. Traits without manifest modes can be used to construct different capabilities, *e.g.,* a trait `Cell` might be used to form both a **locked** `Cell` and a **linear** `Cell` when included in different classes, with different constraints on aliasing of its instances.

As a consequence of this design $\mathcal{K}$ allows the same set of traits to be used to construct classes tailored to different concurrency scenarios, thus contributing to trait-based reuse.

3.2 Dominating and Subordinate Capabilities

Building a data structure from linear capabilities gives *strong encapsulation*: subobjects of the data structure are not aliased from outside. However, linearity imposes a tree-shaped structure on data. Subordinate capabilities instead provide strong encapsulation by forbidding aliases from outside an aggregate to objects within the aggregate. Inside an aggregate, subordinate capabilities may be aliased freely, enabling any graph structure to be expressed.

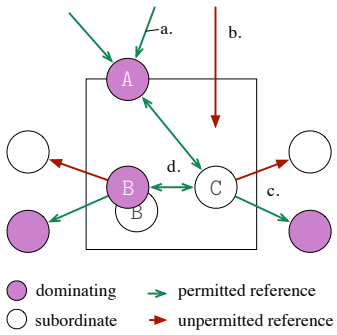
The capabilities **linear**, **thread**, **locked** and **unsafe** are *dominating capabilities* that enclose subordinate capabilities in a statically enforced way. Domination means that all *direct* accesses to objects inside an aggregate from outside are disallowed, making the dominator a single point of entry into an aggregate. As a consequence, any operation on an object inside an aggregate must be triggered by a method call on its dominating capability (directly or indirectly). This means that subordinate objects inherit the concurrency control of their dominator. Subordinate capabilities dominated by a **thread** capability inherit its thread-locality; subordinate capabilities dominated by a **locked** capability enjoys protection of its lock, etc. An implementation of a linked list with **subordinate** links inside a dominating list head guarantees that only a single thread at a time can mutate the links, while still allowing arbitrary internal aliasing inside of the data structure (*e.g.,* doubly-linked, circular).

Figure 1 shows encapsulation in $\mathcal{K}$ from dominating and subordinate capabilities. To enforce the encapsulation of subordinate objects, a subordinate capability (`B` and `C`) may not be returned from or passed outside of its dominating capability (`A`). There is no hierarchical decomposition of the heap (*cf.,* [16]) and no notion of transitive ownership. However, compositions (*cf.* Section 3.3) of dominating and subordinate capabilities (`B`) create nested aggregates, *i.e.,* entire aggregates strongly encapsulated inside another. Pointers to external capabilities must all be to dominating capabilities. Thus, objects inside `B` can refer to `A`, but not to `C`.

3.3 Flat and Nested Composition

As usual in a trait-based system, $\mathcal{K}$ constructs classes by composing traits, or rather capabilities. There are two forms of composition: disjunction ($\oplus$) and conjunction ($\otimes$). If `A` and `B` are capabilities, their disjunction `A` $\oplus$ `B` provides the union of the methods of `A` and `B` and requires the union of their field requirements. Their conjunction `A` $\otimes$ `B` does the same, but is only well-formed if `A` and `B` do not share mutable state which is not protected by concurrency control. This means that `A` $\otimes$ `B` allows `A` and `B` to be used in parallel. Figure 2 shows the composition constraints of disjunction and conjunction pictorially.

We use the term *flat composition* to mean disjunction or conjunction. When employing parametric polymorphism a form of *nested composition* appears. The nested capability `A<B>` exposes that `A` contains zero or more `B`'s at the type level, allowing type-level operations on



a. Disallowed if `A` is **`linear`**. If `A` is **`thread`**, aliases must come from same thread.

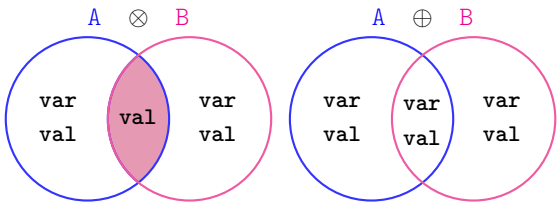
b. References from outside an aggregate to its inside are not permitted.

c. References from inside an aggregate to its outside are permitted if the target is a dominating capability.

d. References inside an aggregate are allowed.

The box encloses the subordinate capabilities of `A`. Note that `B` is a composition of a subordinate *and* a dominating capability (*cf.* Section 3.3), denoted by the two circles. All dominating capabilities have their own boxes (as shown for `A`), *e.g.,* `B` has a box nested inside of `A`'s box, inaccessible to `A` (*cf.,* b.).

Figure 1 Encapsulation: dominating and subordinate capabilities.



Explanation: **`val`** fields are "final", **`var`** fields are mutable. Intersections denote variables shared between (*i.e.,* **`require`**'d by) both capabilities. Types of fields in the filled intersection must be safe, *i.e.,* **`locked`** or **`read`**. Fields of **`subordinate`** type in a conjunction also must not alias.

Figure 2 Permitted sharing of fields and state across two capabilities `A` and `B` in a composite.

the composite capability. (This presentation uses a "dumbed down" version of parametric polymorphism using concrete types in place of polymorphic parameters for simplicity.)

A composite capability inherits all properties and constraints of its sub-capabilities. Linear capabilities must not be aliased at all. Subordinate capabilities must not leak outside their dominator. Consequently, a type which is both **`subordinate`** and **`linear`** is both a dominator (may encapsulate state) and a subordinate (is encapsulated), may not escape its enclosing aggregate and has transfer semantics when assigned (*cf.,* `B` in Figure 1).

Composition affects locking. A disjunction of two locked capabilities `A` ⊕ `B` will be protected by a single lock. A conjunction `A` ⊗ `B` of locked capabilities can use different locks for `A` and `B`, allowing each disjoint part to be locked separately. Furthermore, compositions of **`read`** and **`locked`** capabilities can be mapped to readers–writer locks.

An important invariant in κ is that all aliases are safe with respect to data-races or interference and can be used to the full extent of their types. If an alias can be created, any use of it will not lead to a bad race, either because it employs some kind of locking, because all aliases are read-only, or because the referenced object is exclusive to a particular thread.

4 Creating and Destroying Aliases = Concurrency Control

As aliasing is a prerequisite to sharing objects across possibly parallel computations, creating and destroying aliases is key to enabling parallelism while still guaranteeing race freedom in κ. Alias restrictions allows statically checkable non-interference, *i.e.,* without dynamic concurrency control (*e.g.,* locking). Programs that require objects that are aliased across threads must employ locks or avoid mutation.

Subordinate and thread-local capabilities may only be aliased from within certain contexts. Read, locked and unsafe capabilities have no alias restrictions. Finally, linear capabilities are

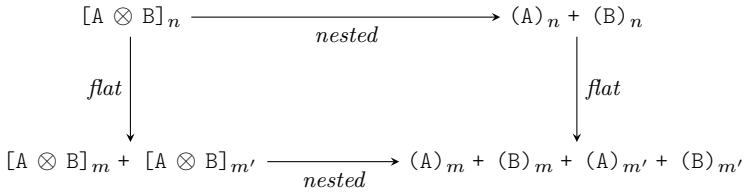


Figure 3 Flat and nested unpacking, using arrays as an analogy. $\texttt{[A} \otimes \texttt{B]}_n$, an array of length n containing composite capabilities $\texttt{A} \otimes \texttt{B}$ can be thought of as a matrix with rows as elements and whose columns are the elements' subparts, `A` and `B`. The matrix can be unpacked by rows (flat) or by columns (nested). Flat unpacking splits the array into subarrays of length m and m' such that $n = m + m'$. Nested unpacking requires that the containing object is not mutable, denoted by turning arrays into tuples, $\texttt{(A)}_m$. These compose in any order producing the same result.

alias-free. The following sections explore how linear types can be manipulated to create and destroy aliases (granting and revoking capabilities) while enjoying non-interference.

4.1 Packing and Unpacking

Conjunctions describe objects constructed from parts that can be manipulated in parallel without internal races. Unpacking breaks an object up into its sub-parts. A variable `c` with a handle to an instance of a class `C`, where **`class`** `C = A` $\otimes$ `B`, can be unpacked into two handles with types `A` and `B` using the `+` operator: **`var`** `a:A + b:B = c`, nullifying `c` in the process.

Unpacking a disjunction is unsafe (and therefore disallowed) since its building blocks can share mutable state not mediated by concurrency control. The dual of unpacking is packing, which re-assembles an object by revoking (nullifying) its sub-capabilities: **`var`** `c:C = a + b`.

The packing and unpacking above is *flat*. Using an array as analogy, flat unpacking takes an array $\texttt{[A]}_n$ with indexes $[0, n)$ and turns it into two disjoint equi-typed sub-arrays with indexes $[0, m)$ and $[m, n)$ where $m \leq n$. $\mathcal{K}$ also allows *nested unpacking*, which in the array analogy means that $\texttt{[A} \otimes \texttt{B]}_n$ can be unpacked into two tuples $\texttt{(A)}_n$ and $\texttt{(B)}_n$ with the same length and indexes. Turning the array into tuples, *i.e.,* immutable arrays of mutable values, is necessary as the aliases could otherwise be used to perform conflicting operations, *e.g.,* updating the `B`-part of element i in one thread and nullifying element i in another thread.

While safe capabilities can always be shared, unpacking allows a linear capability to be split into several aliases that can safely be used concurrently. When restoring the original capability through packing, there may be no residual aliases. We implement this here by preserving linearity in the unpacked capabilities. Figure 3 shows flat and nested unpacking and how they combine and commute. Section 5 shows how unpacking can be used to implement both data parallelism and task parallelism.

In this paper, we only consider packing and unpacking as operations at the level of types: their purpose is to statically guarantee non-interference, not construct new objects from other parts. Thus, packing can be efficiently compiled into an identity check or removed by a compiler provided that handles do not escape the scope in which they were unpacked.

4.2 Bounding Capabilities to the Stack

Linearity is often overly restrictive since it prevents even short-lived aliases that do not break any invariants. To remedy this, $\mathcal{K}$ employs *borrowing* [8]: temporarily relaxing linearity as long as the original capability is not accessible in the same scope, and all aliases are destroyed at the end of the scope. Borrowed capabilities in $\mathcal{K}$ are *stack-bound*, denoted by a

```
class Pair = (linear Fst ⊗ linear Snd) ⊕ linear Swap { var fst:int; var snd:int; }

trait Fst {
  require var fst:int;

  def setFst(i:int) : void {
    this.fst = i;
  }
  def getFst() : int {
    this.fst;
  } }

trait Snd {
  require var snd:int;

  def setSnd(i:int) : void {
    this.snd = i;
  }
  def getSnd() : int {
    this.snd;
  } }

trait Swap {
  require var fst:int;
  require var snd:int;

  def swap() : void {
    var tmp:int = this.fst;
    this.fst = this.snd;
    this.snd = tmp;
  } }
```

Figure 4 A pair class constructed from capabilities, `Fst`, `Snd` and `Swap`.

type wrapper `S()`. For example, `S(linear Cell)` denotes a capability which is identical to the `linear Cell` capability except that it may not be stored in a field, and thus is revoked once the scope exits. κ supports two forms of borrowing:

Forward Borrowing A `linear` capability in a stack variable can be converted into a stack-bound capability for a certain scope, destructively read and then safely reinstated at the end of the scope. This allows *e.g.,* passing a linear capability as an argument to a method, reinstating it on return. In conjunction with the borrowing it may optionally be converted to a `thread`, allowing it to be freely aliased until reinstated.

Reverse Borrowing A method of a `linear` capability may non-destructively read and return a stack-bound alias of a field of `linear` type. This allows linear elements of a data structure to be accessed without removing them, which is safe as long as the capability holding the field is not accessed during borrowing. To prevent multiple reverse borrowings of the same value (which would break linearity), the returned value may not be stored in fields or local variables but must be used immediately, *e.g.,* as an argument to a method call.

Borrowing simplifies programming with linear capabilities as it removes the need to explicitly consume and reinstate values when aliasing is benign, avoiding unnecessary memory writes. See Section 5 for an example of both forward and reverse borrowing in action.

4.3 Forgetting and Recovering Sub-Capabilities

Unpacking a disjunction is unsafe as its building blocks may have direct access to the same state without any concurrency control. As an example, consider the simple `Pair` class created from the capabilities `Fst`, `Snd` and `Swap` shown in Figure 4.

If we could unpack the pair, it would allow `fst` and `snd` to be updated independently. However, this is unsafe in the presence of the `Swap` capability, which accesses both fields. For example, the result of calling `swap()` concurrently with `setFst()` depends on the timing of the threads. A crude solution is simply upcasting `Pair` to `linear Fst ⊗ linear Snd`. This *forgets* the `Swap` capability and enables unpacking—but as a consequence `Swap` is lost forever.

To facilitate recovering a more specific type, κ provides a means to temporarily stash capabilities inside a *jail* which precludes their use except for recovering a composite type:

```
var p:Pair = ...;
var j:J(Pair|Fst ⊗ Snd) + k:(Fst ⊗ Snd) = p; // (1)
var f:Fst + s:Snd = k; // flat unpacking
... // use f and s freely
p = j + (f + s); // flat packing, twice, and getting out of jail (2)
```

At (1), the type of `j`, `J(Pair|Fst ⊗ Snd)`, denotes a jail storing a `Pair` which is unusable (the interface of a jailed capability is empty) until it is unlocked by providing the `Fst ⊗ Snd` capability of the corresponding resource as key. Thus `j` serves as a witness to the existence of the full `Pair` capability, including `Swap`. At (2), we recover `k` from `f` and `s`, nullifying both variables. We use the resulting value to open the jail `j` and store the result in `p`. As for packing, checking whether a key "fits" at run-time (*i.e.,* if `f` and `s` are aliases of the jail) is a simple pointer identity check, which could often be optimised away using escape analysis.

5 Applying Capabilities to the Case Studies in Section 2.1–2.3

Simple Counters. This example demonstrated the problem of distinguishing objects shared across threads from thread-local or unaliased objects objects, and pointed at the trickiness of locking correctly. In $\mathcal{K}$, a counter might be described as a simple trait `Counter`:

```
trait Counter {
  require var cnt : int;
  def inc() : void { this.cnt = this.cnt + 1; }
  def value() : int { return this.cnt; } }
```

To get a capability from the trait, what is missing is to add the mode declaration, which controls aliasing and sharing across threads. Out of the six possible mode annotations, five are allowed for the `Counter` trait:

linear A globally unaliased counter.
thread A thread-local counter. It can be aliased, but aliases cannot cross into other threads.
locked A counter protected by a lock, sharable across threads.
subordinate This type denotes a counter nested inside another object from which it cannot escape. It thus inherits data-race freedom or non-interference of the enclosing object.
unsafe A sharable, unprotected counter that requires the client to perform synchronisation at use-site: `c.inc()` will not compile unless wrapped inside a synchronisation block, which changes the type of `c` from **`unsafe`** to **`locked`**.

Using the mode **`read`** would denote a read-only counter, sharable across threads. Assigning this mode to the trait is rejected by the compiler because of the mutable `cnt` field.

Modes communicate how counters may be aliased: not at all, by a single thread, or across threads. In the latter case modes also communicate how concurrent accesses are made safe: by locks, by only allowing reads (not applicable here), by relying on some containing object or by delegating responsibility to the client.

Differently synchronised counters can be defined almost without code duplication, *e.g.*:

```
class LocalCounter = thread Counter { var cnt:int; }
class SharedCounter = locked Counter { var cnt:int; }
```

Data/Task Parallelism. This example demonstrated the need for reasoning about aliasing in order to determine what parts of an interface can be safely accessed concurrently.

A binary tree can be constructed as the conjunction of capabilities giving access to the left and right subtrees and the current element (full code in the technical report [15]).

```
class Tree<T> = linear Left<T> ⊗ linear Right<T> ⊗ linear Element<T>
```

We employ nesting to show that the tree contains capabilities of type `T`, the type of the element value held by the `Element` capability. The conjunction allows parallel operations on subparts of a tree and requires that parts do not overlap, modulo safe capabilities. Since the

tree type must be treated linearly, the fact that the `Left` and `Right` subtrees do not overlap follows from the requirement that `Left` and `Right` manipulate fields of different names.

To perform data-parallel operations on a tree, we can construct a recursive procedure that takes a tree, splits it into its separate components and operates on them in parallel.

```
def foreach(t:S(Tree<T>), f:T → T) : void {
  var l:S(linear Left<T>) + r:S(linear Right<T>) + e:S(linear Element<T>) = t; // 0
  finish {
    async { foreach(l.getLeft(), f); } // 1
    async { foreach(r.getRight(), f); } // 1
    e.apply(f); } } // 2
```

At (0) the splitting implicitly consumes the original tree capability. At (1) we recurse on the left and right subtrees. At (2) we pass the function argument `f` to the element capability to be performed on its `T`-typed value. For simplicity, we omit the check for whether `l` or `r` is `null`. The implementation requires a tree to be constructed from linear building blocks to guarantee that no parts of the tree are ever shared across multiple threads. `T` does not need to be linear.

This code illustrates both forward and reverse borrowing. The tree argument to `foreach()` is forward borrowed and stack-bound, which is why there is no need to pack `l`, `r` and `e` to recover `t`—`t` is still accessible at the call-site, where it was buried [8] during the call.

Calls to `getLeft()` and `getRight()` return two reversely borrowed linear values (of type `S(Tree<T>)`) which we can pass as arguments to the recursive calls. Hence, all trees manipulated by this code will be stack-bound. If we remove the stack-boundedness, `foreach()` may not update the subtrees in-place, and must recover and return `t` at the end, reminiscent of functional programming. This would cause lines marked (1) to change thus:

```
async { l.setLeft( foreach( l.getLeft(), f ) ); }
async { l.setRight( foreach( l.getRight(), f ) ); }
```

which allows *replacing* the tree as opposed to updating it, plus a return: `return l + r + e`.

We may extend the `Tree` type with a disjunction on a capability `Visit` which provides a read-only view of the entire tree. Elements may not be swapped for other elements, but modified if `T` allows it. This allows multiple threads to access the same tree in parallel provided that `Left`, `Right` and `Element` are temporarily forgotten.

```
class Tree<T> = read Visit<T> ⊕ (linear Left<T> ⊗ linear Right<T> ⊗ linear Element<T>)
```

Let the type of our tree be `Tree<A ⊗ B>` for linear capabilities `A` and `B`. Turning this capability into `Visit<A ⊗ B>` is possible by forgetting every other capability in the tree type. While read-only capabilities can be aliased freely, creating multiple aliases typed `Visit<A ⊗ B>` would provide multiple paths to supposedly linear `A ⊗ B` capabilities. Composition must thus adhere to all alias restrictions in the composite capability, just like flat composition. Therefore, `Visit<A ⊗ B>` is a linear capability. Unpacking however allows us to turn `Visit<A ⊗ B>` into two handles typed `Visit<A>` and `Visit<B>`, which preserves linearity across all paths. This allows us to specify a task-parallel operation which implements column-based access:

```
def map(t:S(Tree<A⊗B>), f:S(A)→void, g:S(B)→void) : void{
  var ta:S(read Visit<A>) + tb:S(read Visit<B>) = t; // 3
  finish {
    async { ta.preorder(f); } // 4
    async { tb.preorder(g); } } }// 4
```

In this code we create two immutable views of the spine of the tree using `Visit` and then proceed to apply `f` and `g` to all elements of the tree in parallel. At (3) the rest of the capabilities of `Tree` are forgotten. If we wanted to restore them after the parallel operations we would jail them at (3) and restore them after (4).

While the data-parallel version is more scalable than the task-parallel version, there may be cases when the latter is preferred. Further, their combination is possible in either order—apply `f` and `g` in parallel to each element at (2) above, or start by unpacking the tree into multiple immutable trees and then process the sub-elements in parallel in each tree, equivalent to calling a version of `foreach` instead of `preorder` at (4) (*cf.,* Figure 3).

Vector vs. ArrayList in Java. This example demonstrated that building synchronisation into a data structure can cause too much overhead and destroy parallelism. In κ, a list might be described using capabilities (full code in technical report, [15]):

- `Add_Del` for adding and removing elements
- `Get` for looking up elements

`Add_Del` might be split into two capabilities allowing for more flexibility, for example granting a client only the ability to add elements but not delete them. As the two capabilities operate on some shared state (the links), their combination must be a disjunction: `Add_Del` $\oplus$ `Get`.

To express the difference between the array list and vector, we would write

```
class ArrayList = unsafe Add_Del ⊕ unsafe Get // Needs external synchronisation
class Vector = locked Add_Del ⊕ locked Get // Has synchronisation built in
```

Specifying use of readers–writer locks to access an object is straightforward and allows sharing a list across threads for reading, causing concurrent write operations to block:

```
class ArrayList = unsafe Add_Del ⊕ read Get
class Vector = locked Add_Del ⊕ read Get
```

The use of **`unsafe`** in the definition of the array list class pushes the synchronisation from within the called methods to the outside, *e.g.,* calling `list.add(element)` we must first take a (write-)lock on `list`. Requiring external synchronisation also allows acquiring, holding and releasing a lock once to perform several operations, like an iteration, without fear of interleaving accesses from elsewhere.

The type **`thread`** `Add_Del` $\oplus$ **`read`** `Get` denotes a list confined to its creating thread. The type **`linear`** `Add_Del` $\oplus$ **`read`** `Get` denotes a list that can mediate between being mutated from one alias or read-only from several aliases. This type is similar to a readers–writer lock, except relying on alias restrictions instead of locks (*cf.,* [9]), removing locking overhead. The ability to reuse traits for different concurrency scenarios is an important contribution of κ.

Concluding Remarks for Section 3–5

Linear and thread-local capabilities give *non-interference* by restricting aliases to a single thread. Locked and unsafe capabilities can be shared across threads and employ locks at declaration-site or at use-site to *avoid data-races*. Read capabilities can be shared across threads and do not allow causing or directly witnessing mutation. When a read capability is extracted from a linear composite, no mutating aliases exist, guaranteeing *non-interference*. When extracted from a locked composite, locks are used to guarantee *data-race freedom*.

The assignment of modes to traits at inclusion site allow a single definition to be reused across multiple concurrency scenarios. Composition captures how different parts of an object's interface interact and defines the safe aliasing of an object.

Subordinate capabilities inherit the protection of their enclosing dominating capabilities. Thus, operations on encapsulated objects are atomic in κ, in the sense that all side-effects of a method call on an aggregate are made visible to other threads atomically. Operating atomically on several objects which are not encapsulated in the same aggregate is possible by

$$
\begin{array}{rcll}
P & ::= & Cds\ Tds\ e & \textit{(Program)} \\
Cd & ::= & \texttt{class}\ \texttt{C} = \texttt{K}\ \{\ Fds\ \} & \textit{(Class definition)} \\
Fd & ::= & mod\ f : \texttt{t} & \textit{(Field definition)} \\
mod & ::= & \texttt{var} \mid \texttt{val} & \textit{(Mutable and immutable fields)} \\
\texttt{K} & ::= & k\ \texttt{T} \mid k\ \mathbf{I} \mid \texttt{K}\langle\texttt{K}\rangle \mid (\texttt{K} \odot \texttt{K}) & \textit{(Capabilities and composition)} \\
\odot & ::= & \otimes \mid \oplus & \textit{(Conjunction and disjunction)} \\
Td & ::= & k\ \texttt{trait}\ \texttt{T}\langle t\rangle\ \{Rs\ Mds\ \} \mid \texttt{trait}\ \texttt{T}\langle t\rangle\ \{Rs\ Mds\ \} & \textit{(Trait definition)} \\
R & ::= & \texttt{require}\ Fd & \textit{(Field requirement)} \\
Md & ::= & \texttt{def}\ m(x : t) : t\ \{\ e\ \} & \textit{(Method definition)} \\
\texttt{e} & ::= & v \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{pack}\ x = y\ +\ z\ \texttt{in}\ e \mid \texttt{unpack}\ x\ +\ y = z\ \texttt{in}\ e \mid x.m(e) \mid x \mid x.f & \\
 & \mid & x.f = e \mid \texttt{new}\ \texttt{C} \mid \texttt{consume}\ x \mid \texttt{consume}\ x.f \mid (t)\ e \mid \texttt{sync}\ x\ \texttt{as}\ y\ \{\ e\ \}; e & \\
 & \mid & \texttt{bound}\ x\ \{\ e\ \}; e \mid \texttt{finish}\ \{\ \texttt{async}\ \{\ e\ \}\ \texttt{async}\ \{\ e\ \}\ \}; e & \textit{(Expression)} \\
v & ::= & \texttt{null} & \textit{(Literal)} \\
t & ::= & \texttt{K} \mid \texttt{C} \mid B(\texttt{K}) & \textit{(Type)} \\
B & ::= & \texttt{J}_{\texttt{K}} \mid \texttt{S} & \textit{(“Boxed” types, i.e., jailed or stack-bound)} \\
k & ::= & \texttt{linear} \mid \texttt{locked} \mid \texttt{read} \mid \texttt{safe} \mid \texttt{subordinate} \mid \texttt{thread} \mid \texttt{unsafe} & \textit{(Modes)}
\end{array}
$$

Figure 5 Syntax of κ. T is a trait name; $\mathbf{I}$ is the incapability; C is a class name; m is a method name; f is a field name; x, y are variable names, including **this**. $Ds ::= D_1, \ldots, D_n$ for $D \in \{Cd, Td, Fd, R, Md\}$.

locking them together using nested synchronisation (for **unsafe** capabilities) or by structuring a call-chain on **locked** capabilities.

Invariantly, all well-typed aliases can coexist without risking data-races. The type system guarantees that all accesses to an object will either be exclusive or only perform operations that cannot clash with any other possible concurrent operations to the same object.

6 Formalising κ

We formalise the static semantics of κ. We define a flattening translation into a language without traits, κ_{F}, whose static and dynamic semantics is found in the accompanying technical report [15]. κ_{F} is a simple object-oriented language with structured parallelism and locking, that uses classes and interfaces which are oblivious to the existence of κ capabilities. The translation from κ to κ_{F} inserts locking and unlocking operations when translating **locked** capabilities and conjunctions of **locked** and **read** capabilities. The locks are reentrant readers–writer locks controlling access to parts of objects. Other locking schemes are possible.

The syntax of κ is shown in Figure 5. We make a few simplifications, none of which are critical for the soundness of the approach:

1. We use let-bindings and explicit pack/unpack constructs. Targets of method calls must be stack variables. Aliasing stack-bounds requires a method-call indirection.
2. We consider finish/async parallelism rather than unstructured creation of threads.
3. Classes only contain fields and no methods.
4. We omit the treatment of constructors. Fields are initialised with **null** on instantiation.
5. We use objects to model higher-order functions and omit these from the formalism.
6. Only a single method parameter and a single nested type are supported.

We introduce a **safe** capability, which abstracts **read** and **locked** to allow mode subtyping. The **safe** mode is only allowed in types, not in declarations. The *incapability* type $\mathbf{I}$ does not contain any fields or methods and simply allows holding a reference to an object.

$\boxed{\vdash P : t \quad \vdash Td \quad \Gamma \vdash Td \quad \vdash Cd}$ *(Well-formedness top-level declarations)*

WF-PROGRAM
$$\frac{\forall\ Cd \in Cds\ .\ \vdash Cd \quad \forall\ Td \in Tds\ .\ \vdash Td \quad \epsilon \vdash e : t}{\vdash Cds\ Tds\ e : t}$$

WF-T-TRAIT
$$\frac{\vdash \mathbf{subord}\ \mathbf{trait}\ \mathtt{T}\langle t\rangle\ \{\ Rs\ Mds\ \}}{\vdash \mathbf{trait}\ \mathtt{T}\langle t\rangle\ \{\ Rs\ Mds\ \}}$$

WF-T-TRAIT-MFST
$$\frac{\rho : t \vdash k\ \mathbf{trait}\ \mathtt{T}\ \{\ Rs\ Mds\ \}}{\vdash k\ \mathbf{trait}\ \mathtt{T}\langle t\rangle\ \{\ Rs\ Mds\ \}}$$

WF-T-INNARDS
$$\frac{k \neq \mathbf{safe} \quad \Gamma, \mathbf{this} : k\,\mathtt{T} \vdash Rs \quad \Gamma, \mathbf{this} : k\,\mathtt{T} \vdash Mds}{\Gamma \vdash k\ \mathbf{trait}\ \mathtt{T}\ \{\ Rs\ Mds\ \}}$$

WF-CLASS
$$\frac{\vdash \mathtt{K} \quad \forall\ Fd \in Fds\ .\ \mathbf{this} : \mathtt{K} \vdash Fd \quad \forall\ \mathbf{var}\ f : t_1 \in \mathbf{fields}\,(\mathtt{K})\ .\ \exists\ \mathbf{var}\ f : t_2 \in Fds\ .\ t_2 \equiv t_1 \quad \forall\ \mathbf{val}\ f : t_1 \in \mathbf{fields}\,(\mathtt{K})\ .\ \exists\ \mathbf{var}\ f : t_2 \in Fds\ .\ t_2 <: t_1}{\vdash \mathbf{class}\ C = \mathtt{K}\ \{\ Fds\ \}}$$

$\boxed{\Gamma \vdash R, Rs \quad \Gamma \vdash Mds, Md \quad \Gamma \vdash Fd \quad \vdash \Gamma}$ *(Well-formed body parts)*

WF-REQ-FD
$$\frac{\Gamma \vdash Fd}{\Gamma \vdash \mathbf{require}\ Fd}$$

WF-REQ-FDS
$$\frac{\Gamma \vdash R \quad \Gamma \vdash Rs}{\Gamma \vdash Rs, R}$$

WF-MDS
$$\frac{\Gamma \vdash Md \quad \Gamma \vdash Mds}{\Gamma \vdash Mds, Md}$$

WF-M-TRAIT
$$\frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \mathbf{def}\ m(x : t_1) : t_2\ \{\ e\ \}}$$

WF-FD-NST
$$\frac{\vdash \Gamma \quad \Gamma(\rho) = \mathtt{K}}{\Gamma \vdash \mathbf{val}\ f : \mathtt{K}}$$

WF-FD
$$\frac{\Gamma(\mathbf{this}) = \mathtt{K} \quad \vdash t \quad t \neq \mathbf{S}(\_) \quad \vdash \mathtt{K} \quad \mathbf{thread}(t) \Rightarrow \mathbf{thread}(\mathtt{K}) \quad \mathbf{read}(\mathtt{K}) \Rightarrow (mod \equiv \mathbf{val} \wedge \mathbf{safe}(t))}{\Gamma \vdash mod\ f : t}$$

ENV-E
$$\frac{}{\vdash \epsilon}$$

ENV-VAR
$$\frac{\vdash \Gamma \quad \vdash t \quad x \notin \mathbf{dom}(\Gamma)}{\vdash \Gamma, x : t}$$

Figure 6 Well-formed declarations. $\Gamma ::= \epsilon \mid \Gamma, x : t$, ($x$ incl. ρ).

Our main technical result is the proof that a κ_{F} program translated from a well-typed κ program enjoys safe aliasing and strong encapsulation (*cf.* Section 7.2) in a way that implies thread-safety (*cf.* Section 7.3). We verify our definition of thread-safety by proving that it implies data-race freedom and, when certain capabilities are excluded, also non-interference (*cf.* Section 7.3).

Helper Predicates and Functions. The functions **fields**, **vals**, **vars** and **msigs** return a map from names to types or method signatures. We use helper predicates of the form k(K) to assess whether a capability K has mode k. The predicates **linear**, **subord**(inate) and **unsafe** hold if there exists *at least one* sub-capability in K of that mode. The predicates **read**(K) and **encaps**(K) hold if *all* sub-capabilities in K are **read** or **subordinate**, respectively. **locked**(K) holds if one or more sub-capabilities are locked, and the remainder **safe**.

6.1 Well-Formed κ Programs (Figure 6)

A well-formed program consists of classes, traits, and an initial expression (WF-PROGRAM). Traits without manifest mode are type-checked as if they were subordinate (WF-T-TRAIT). To reduce the number of rules, we require all traits to have exactly one nested capability (a concrete type "parameter"), and use T as shorthand for T<**I**>, where **I** is the empty capability. A trait is well-formed if its field requirements and methods are well-typed given the self-type of the current trait and the nested type. The latter is tracked by the special variable ρ which

$\boxed{\vdash t}$ *(well-formedness of types)* $\boxed{Fd_1 \odot Fd_2}$ *(sharing fields across traits)*

T-CLASS
$$\frac{\textbf{class}\ C = \mathtt{K}\,\{\ \_\ \} \in P}{\vdash C}$$

T-BOXED
$$\frac{\vdash \mathtt{K}}{\vdash B(\mathtt{K})}$$

T-NESTING
$$\frac{\vdash \mathtt{K}_1 \quad \vdash \mathtt{K}_2}{\vdash \mathtt{K}_1\langle\mathtt{K}_2\rangle}$$

C-SHARABLE
$$\frac{\textbf{safe}(t) \vee \textbf{unsafe}(t)}{\textbf{val}\, f : t \otimes \textbf{val}\, f : t}$$

T-TRAIT-MFST
$$\frac{k\ \textbf{trait}\ \mathtt{T}\langle\mathtt{K}\rangle\ \{\ \_\_\_\ \} \in P}{\vdash k\,\mathtt{T}}$$

T-TRAIT
$$\frac{k \equiv \textbf{read} \Rightarrow \textbf{this} : \textbf{read}\,\mathtt{T} \vdash Rs \quad \textbf{trait}\ \mathtt{T}\langle\mathtt{K}\rangle\ \{\ Rs\ \_\ \} \in P}{\vdash k\,\mathtt{T}}$$

C-VAR-VAL
$$\frac{t_1 <: t_2}{\textbf{var}\, f : t_1 \oplus \textbf{val}\, f : t_2}$$

C-VAL-VAL
$$\frac{\vdash \mathtt{K}_1 \otimes \mathtt{K}_2 \quad \odot\ =\ \otimes \Rightarrow \neg\,\textbf{subord}(\mathtt{K}_1 \otimes \mathtt{K}_2)}{\textbf{val}\, f : \mathtt{K}_1 \odot \textbf{val}\, f : \mathtt{K}_2}$$

T-COMPOSITION
$$\frac{\vdash \mathtt{K}_1 \qquad \vdash \mathtt{K}_2 \quad \forall\ Fd_1 \in \textbf{fields}\,(\mathtt{K}_1), Fd_2 \in \textbf{fields}\,(\mathtt{K}_2)\ .\ Fd_1 \odot Fd_2 \quad \textbf{wfRegions}\,(\mathtt{K}_1, \mathtt{K}_2) \qquad \textbf{wfRegions}\,(\mathtt{K}_2, \mathtt{K}_1)}{\vdash \mathtt{K}_1 \odot \mathtt{K}_2}$$

T-I
$$\frac{}{\vdash k\,\mathbf{I}}$$

C-DISJOINT
$$\frac{f_1 \neq f_2}{mod_1\, f_1 : t_1 \odot mod_2\, f_2 : t_2}$$

T-REGIONS
$$\frac{\forall\ \mathtt{K}_1 \otimes \mathtt{K}_2 \in \mathtt{K}\ .\ mod_1\, f_1 : t_1 \in \textbf{fields}\,(\mathtt{K}_1) \wedge mod_2\, f_2 : t_2 \in \textbf{fields}\,(\mathtt{K}_2) \wedge f_1 \neq f_2 \wedge \textbf{subord}(t_1) \wedge \textbf{subord}(t_2) \Rightarrow \neg\,(f_1 \in \textbf{fields}\,(\mathtt{K}') \wedge f_2 \in \textbf{fields}\,(\mathtt{K}'))}{\textbf{wfRegions}\,(\mathtt{K}, \mathtt{K}')}$$

C-DISJUNCTION
$$\frac{}{mod_1\, f : t \oplus mod_2\, f : t}$$

Figure 7 Well-formed types. Conjunctions and disjunctions of traits are governed by the rules for overlapping fields. For simplicity, we omit (C-VAL-VAR) which is isomorphic with (C-VAR-VAL).

may not appear anywhere in the program source (WF-T-TRAIT-MFST). Fields are either mutable (`var`) or stable (`val`). We assume that names of classes and traits are unique in a program and the names of fields and methods are unique in classes and traits.

A well-formed class consists of well-typed `var` fields that satisfy the requirements from its traits, and a defined equivalence to a well-formed composite capability. We allow covariance for `val` fields (WF-CLASS). Only immutable fields holding `safe` capabilities are allowed in `read` capabilities (WF-REQ-*,WF-FD), unless the type of the field is exposed through nesting (WF-FD-NST). Fields may not store stack-bound capabilities and fields holding thread-local values are only allowed if the containing object is also thread-local (WF-FD).

6.2 Well-Formed Types (Figure 7)

Capabilities corresponding to traits with manifest modes are trivially well-formed (T-TRAIT-MFST). Traits without a manifest mode can be given any mode (T-TRAIT). Well-formed `read` capabilities may only contain `safe` val fields. The empty capability **I** can be given any mode (T-I). Composing capabilities with **I** thus affects the mode of the composite, but not the interface (*cf.,* Section 6.3).

Two well-formed capabilities can form a nested capability type (T-NESTING). A composite capability is well-formed if its sub-capabilities are well-formed and their shared fields are composable (T-COMPOSITION). We also require that two subordinate fields appearing on opposing sides of a conjunction $\mathtt{K}_1 \otimes \mathtt{K}_2$ are not both accessible from some other trait $\mathtt{K}'$ in the same composite (T-REGIONS). Such a field would act as a channel that could be used to share subordinate state across the supposedly disjoint representations of $\mathtt{K}_1$ and $\mathtt{K}_2$.

The rules of the form $Fd_1 \odot Fd_2$ govern field overlaps between capabilities in a composite, where $\odot \in \{\otimes, \oplus\}$ denotes the composition of the capabilities containing the fields (*cf.*, Figure 2). Disjunctions may overlap freely (C-DISJUNCTION). Disjoint fields do not overlap (C-DISJOINT). If a field appearing on both sides of a composition is mutable on one side and immutable on the other, the mutable field's type must be more precise (C-VAR-VAL). An immutable field may appear on both sides of a composition only if its type is safe or unsafe (C-SHARABLE) or if the fields have types whose conjunction is well-formed (C-VAL-VAL). If the sharing capabilities are conjunctive, the field must not be subordinate.

6.3 Type Equivalence, Packing and Subtyping (Figure 8)

Class names are aliases for composite capabilities (T-EQ-CLASS-TRAIT). The order of the operands in composition of *a single kind* does not matter (T-EQ-COMMUTATIVE) and (T-EQ-ASSOCIATIVE). Equivalent types in jail or bound to the stack are still equivalent (T-EQ-BOXED). A **read** with a nested conjunction can be unpacked into two **read** capabilities with nested capabilities from the unpacked conjunction (T-EQ-NESTING). Incapabilities can be added and removed from a type as long as modes are preserved (T-EQ-I).

Jailing allows creating a conjunction from a disjunction (T-JAIL). The full capability that can be unlocked from the jail is written as a subscript K. Jailing requires that all modes on the composite are preserved by the extracted capability, modulo the rules $k_{<:}$ for unpacking to be sound. For example, **locked** A $\oplus$ **subord** B denotes a (partially) subordinate capability that is strongly encapsulated inside an aggregate. If we are allowed to forget the subordinate mode of the type, the locked sub-capability could be leaked. We employ (T-EQ-I) to this end. For example, we can compose **subord** **I** with **locked** A $\oplus$ **subord** B and then apply (T-JAIL) to extract **locked** A $\oplus$ **subord** **I** which satisfies mode preservation.

Subtyping is structural on capabilities. Subtyping must preserve modes, or encapsulation, domination or exclusivity could be lost. The rules (T-SUB-*) allow **locked** and **read** to be abstracted by the **safe** mode.

6.4 Well-Typed Expressions (Figure 9)

Packing & Unpacking. The rules (E-PACK) and (E-UNPACK) govern the packing and unpacking of capabilities. They rely on the rules (T-JAIL) and (T-PACK) in Figure 8 which allow introducing aliases to discrete capabilities of a conjunction and turning a disjunction into a conjunction by jailing the overlapping parts.

Linearity. **linear** capabilities are destructively read to maintain alias freedom and allow ownership transfer (E-CONS-VAR,E-CONS-FD). Method calls do not destroy **linear** receivers as an object's **this** cannot be consumed. Thus, linear capabilities are externally unique [17].

Finish–Async and Sync. Parallelism in $\mathcal{K}$ is modelled using a scoped finish/async construct (abbreviated as **f** and **a** respectively) (E-FJ). A finish–async forks two parallel computations and waits until they have both completed. Forking a larger number of threads can be simulated using nested finish/async blocks. For simplicity we do not allow async blocks outside of a finish block, but extending the system to support unstructured parallelism or active objects is possible. We employ a frame rule that guarantees that no variable is visible in both asyncs, and that subordinate objects are only accessible to the first of the asyncs, (FRAME). This models the current thread running the first async (with access to the current **this**), and another thread running the second async block.

$\boxed{t_1 \equiv t_2}$ *(type equivalence)*

T-EQ-CLASS-TRAIT
$$\frac{\textbf{class}\ C = \mathtt{K}\,\{\ \_\ \} \in P}{C \equiv \mathtt{K}}$$

T-EQ-COMMUTATIVE
$$\frac{}{\mathtt{K}_1 \odot \mathtt{K}_2 \equiv \mathtt{K}_2 \odot \mathtt{K}_1}$$

T-EQ-ASSOCIATIVE
$$\frac{}{(\mathtt{K}_1 \odot \mathtt{K}_2) \odot \mathtt{K}_3 \equiv \mathtt{K}_1 \odot (\mathtt{K}_2 \odot \mathtt{K}_3)}$$

T-EQ-BOXED
$$\frac{\mathtt{K}_1 \equiv \mathtt{K}_2}{B(\mathtt{K}_1) \equiv B(\mathtt{K}_2)}$$

T-EQ-NESTING
$$\frac{\textbf{read}(\mathtt{K}_1)}{\mathtt{K}_1\langle \mathtt{K}_2 \otimes \mathtt{K}_3\rangle \equiv \mathtt{K}_1\langle \mathtt{K}_2\rangle \otimes \mathtt{K}_1\langle \mathtt{K}_3\rangle}$$

T-EQ-I
$$\frac{k(\mathtt{K})}{\mathtt{K} \odot k\,\mathbf{I} \equiv \mathtt{K}}$$

T-EQ-TRANS
$$\frac{\mathtt{K}_1 \equiv \mathtt{K}_2 \qquad \mathtt{K}_2 \equiv \mathtt{K}_3}{\mathtt{K}_1 \equiv \mathtt{K}_3}$$

$\boxed{t_1 \rightleftharpoons t_2 \otimes t_3}$ *(packing and unpacking)*

T-PACK
$$\frac{\mathtt{K}_1 \equiv \mathtt{K}_2 \otimes \mathtt{K}_3 \qquad \textbf{subord}(\mathtt{K}_2) \Leftrightarrow \textbf{subord}(\mathtt{K}_3)}{\mathtt{K}_1 \rightleftharpoons \mathtt{K}_2 \otimes \mathtt{K}_3}$$

T-JAIL
$$\frac{\mathtt{K}_1 \equiv \mathtt{K}_2 \oplus \mathtt{K}_3 \qquad \forall\ k\ .\ k(\mathtt{K}_1) \Rightarrow k(\mathtt{K}_3)}{\mathtt{K}_1 \rightleftharpoons \mathbf{J}_{\mathtt{K}_1}(\mathtt{K}_2) \otimes \mathtt{K}_3}$$

T-PACK-BOUND
$$\frac{\mathtt{K}_1 \rightleftharpoons \mathtt{K}_2 \otimes \mathtt{K}_3}{\mathbf{S}(\mathtt{K}_1) \rightleftharpoons \mathbf{S}(\mathtt{K}_2) \otimes \mathbf{S}(\mathtt{K}_3)}$$

$\boxed{t_1 <: t_2 \quad k_{<:}(\mathtt{K})}$ *(subtyping and "submoding")*

T-SUB-STRUCTURAL
$$\frac{\forall\ k\ .\ k(\mathtt{K}_1 \odot \mathtt{K}_2) \Rightarrow k_{<:}(\mathtt{K}_1)}{\mathtt{K}_1 \odot \mathtt{K}_2 <: \mathtt{K}_1}$$

T-SUB-BOXED
$$\frac{\mathtt{K}_1 <: \mathtt{K}_2}{B(\mathtt{K}_1) <: B(\mathtt{K}_2)}$$

T-SUB-EQ
$$\frac{t_1 \equiv t_2 \qquad t_2 <: t_3}{t_1 <: t_3}$$

T-SUB-ID
$$\frac{}{t <: t}$$

T-SUB-K
$$\frac{k(\mathtt{K})}{k_{<:}(\mathtt{K})}$$

T-SUB-RD
$$\frac{\textbf{safe}(\mathtt{K})}{\textbf{read}_{<:}(\mathtt{K})}$$

T-SUB-LOCK
$$\frac{\textbf{safe}(\mathtt{K})}{\textbf{locked}_{<:}(\mathtt{K})}$$

Figure 8 Type equivalence, packing/unpacking, subtyping.

FRAME
$$\frac{\textbf{dom}\,(\Gamma_2) \cap \textbf{dom}\,(\Gamma_3) \equiv \emptyset \qquad \nexists\ x : t \in \Gamma_3\ .\ (\textbf{subord}(t) \vee \textbf{thread}(t)) \qquad \forall\ x\ .\ (\Gamma_2(x) = t \Rightarrow \Gamma_1(x) = t) \wedge (\Gamma_3(x) = t \Rightarrow \Gamma_1(x) = t)}{\Gamma_1 = \Gamma_2 + \Gamma_3}$$

The `sync` keyword temporarily converts an `unsafe` (and therefore unusable) capability into a `locked` capability, acquiring and releasing locks at the entry and exit of e_1 (E-SYNC).

Borrowing. Forward borrowing allows turning capabilities into stack-bound capabilities non-destructively, possibly relaxing a `linear` to a `thread` and allows splitting reads (E-FORWARD). The helper predicate **boundable**($\mathtt{K}_1$, $\mathtt{K}_2$) holds in either of three cases:

1. $\mathtt{K}_1 = \mathtt{K}_2$
2. $\mathtt{K}_1[\texttt{linear} \mapsto \texttt{thread}] = \mathtt{K}_2$ (relaxing linearity to thread-affinity)
3. $\mathtt{K}_1 = \mathtt{K}_2 \odot \mathtt{K}_3$ s.t. $\mathtt{K}_3$ is neither `subordinate` nor `thread`, and all capabilities in $\mathtt{K}_2$ are `read`.

The last case allows relaxing alias restrictions for stack-bound `read` capabilities which enables mediating from single writer to multiple readers across multiple threads without dynamic concurrency control for a clearly defined scope.

Reverse borrowing allows non-destructively reading a `linear` capability into a stack-bound value (E-REVERSE). Since only one value can be returned by a method, multiple reverse borrowing of the same field in the same method is innocuous. Non-linear capabilities never need to be reverse borrowed as they can always be returned normally without consuming.

Self Typing. Modulo traits with manifest modes, `this` inside a trait is always subordinate (WF-T-TRAIT). This reflects the fact that on the inside of a capability, exclusive access of a

$\boxed{\Gamma \vdash e : t}$ *(expression typing)*

E-UPCAST
$$\frac{\Gamma \vdash e : t_2 \qquad t_2 <: t_1}{\Gamma \vdash (t_1)e : t_1}$$

E-NEW
$$\frac{\vdash \Gamma \qquad \textbf{class}\ C = \mathtt{K}\{\_\} \in P}{\Gamma \vdash \textbf{new}\ C : \mathtt{K}}$$

E-NULL
$$\frac{\vdash \Gamma \qquad \vdash t}{\Gamma \vdash \textbf{null} : t}$$

E-LET
$$\frac{\Gamma \vdash e_1 : t_1 \qquad t_1 \neq \mathbf{S}(\_) \qquad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : t_2}$$

E-UNPACK
$$\frac{\Gamma \vdash z : t_1 \qquad t_1 \rightleftharpoons t_2 \otimes t_3 \qquad \Gamma, x : t_2, y : t_3 \vdash e : t_4}{\Gamma \vdash \textbf{unpack}\ x + y = z\ \textbf{in}\ e : t_4}$$

E-PACK
$$\frac{\Gamma \vdash y : t_1 \qquad \Gamma \vdash z : t_2 \qquad \textbf{linear}(t_1) \qquad \textbf{linear}(t_2) \qquad t_3 \rightleftharpoons t_1 \otimes t_2 \qquad \Gamma, x : t_3 \vdash e : t_4}{\Gamma \vdash \textbf{pack}\ x = y + z\ \textbf{in}\ e : t_4}$$

E-SELECT
$$\frac{\Gamma \vdash \textbf{this} : t_1 \qquad \textbf{fields}\,(t_1)(f) = t_2 \qquad \neg\,\textbf{linear}(t_2)}{\Gamma \vdash \textbf{this}.f : t_2}$$

E-CONS-FD
$$\frac{\Gamma \vdash \textbf{this} : t_1 \qquad \textbf{vars}\,(t_1)(f) = t_2}{\Gamma \vdash \textbf{consume}\,\textbf{this}.f : t_2}$$

E-UPDATE
$$\frac{\Gamma \vdash \textbf{this} : t_1 \qquad \textbf{vars}\,(t_1)(f) = t_2 \qquad \Gamma \vdash e : t_2}{\Gamma \vdash \textbf{this}.f = e : t_2}$$

E-VAR
$$\frac{\vdash \Gamma \qquad \Gamma(x) = t \qquad \neg\,\textbf{linear}(t)}{\Gamma \vdash x : t}$$

E-CONS-VAR
$$\frac{\vdash \Gamma \qquad x \neq \textbf{this} \qquad \Gamma(x) = t}{\Gamma \vdash \textbf{consume}\ x : t}$$

E-CALL
$$\frac{\textbf{linear}(t_1) \Rightarrow x \notin \textbf{freeVars}\,(e) \qquad \Gamma(x) = t_1 \qquad \neg\,\textbf{unsafe}(t_1) \qquad \textbf{msigs}\,(t_1)(m) = z : t_2 \to t_3 \qquad \Gamma \vdash e : t_2 \qquad (\textbf{subord}(t_2) \vee \textbf{subord}(t_3)) \Rightarrow \textbf{encaps}\,(t_1) \vee x \equiv \textbf{this}}{\Gamma \vdash x.m(e) : t_3}$$

E-FJ
$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma \vdash e : t \qquad \Gamma_1 \vdash e_1 : \_ \qquad \Gamma_2 \vdash e_2 : \_}{\Gamma \vdash \mathbf{f}\,\{\ \mathbf{a}\ \{\ e_1\ \}\ \mathbf{a}\ \{\ e_2\ \}\ \};\, e : t}$$

E-SYNC
$$\frac{\Gamma \vdash x : \textbf{unsafe}\,\mathtt{T} \qquad \Gamma, y : \mathbf{S}(\textbf{locked}\,\mathtt{T}) \vdash e_1 : \_ \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash \textbf{sync}\ x\ \textbf{as}\ y\ \{\ e_1\ \};\, e_2 : t}$$

E-REVERSE
$$\frac{\Gamma \vdash \textbf{this} : t \qquad \textbf{linear}(t) \qquad \textbf{fields}\,(t)(f) = \mathtt{K}}{\Gamma \vdash \textbf{this}.f : \mathbf{S}(\mathtt{K})}$$

E-FORWARD
$$\frac{\textbf{boundable}\,(\mathtt{K}_1, \mathtt{K}_2) \qquad \Gamma, x : \mathbf{S}(\mathtt{K}_2) \vdash e_1 : \_ \qquad \Gamma, x : \mathtt{K}_1 \vdash e_2 : t}{\Gamma, x : \mathtt{K}_1 \vdash \textbf{bound}\ x\ \{\ e_1\ \};\, e_2 : t}$$

Figure 9 Typing of expressions. Note that all fields are "private".

single thread is already guaranteed (*e.g.,* because the accessing thread was forced to acquire a lock to enter the object, or place a linear entry point in a stack variable, which is analogous).

Viewing `this` as subordinate allows an object to be aliased freely from *inside* its own enclosure, including objects of linear capabilities. Thus, linear capabilities in κ are externally unique [17]. Traits with manifest modes have more information about themselves internally.

7 Meta-Theoretic Evaluation

In this section, we describe the key invariants of our capabilities in a well-typed κ program. Due to lack of space, proofs have been relegated to a technical report [15]. This section aims to explain the key properties, and sketching why they hold.

7.1 κ_{F} and the Dynamic Semantics of κ Programs

To execute κ programs, they are translated into a simpler language, κ_{F}, with classes and interfaces without traits or capabilities. The semantics of κ_{F} is straightforward, and only the most relevant details are presented here.

Flattening of traits [39] is performed similar to other trait-based languages: A class is translated by copying the methods from its traits, traits are translated into interfaces with the equivalent signature. For each composition $\mathtt{K}_1 \odot \mathtt{K}_2$, an interface is created extending the interfaces corresponding to $\mathtt{K}_1$ and $\mathtt{K}_2$. This preserves the same subtyping rules as in $\mathcal{K}$. For each $\mathcal{K}$-capability in a translated program, there is a single corresponding interface. Because of this one-to-one mapping, we can easily recover $\mathcal{K}$ types from a $\mathcal{K}_\mathrm{F}$ program, which we use extensively in our proofs.

From field overlaps, we infer a set of regions for each class, and insert lock and unlock instructions at the start and end of methods to acquire and release the lock for the region touched by the method. Read locks are acquired in methods in read capabilities if they overlap with a locked or unsafe capability. Methods in locked capabilities acquire write locks. All locks are reentrant. Well-formedness of $\mathcal{K}_\mathrm{F}$ configurations requires that no two threads can hold the same writer lock, and that all locks held by a thread are also taken in the objects themselves. This assures mutual exclusion in all parts of a program that use locks.

A well-formed $\mathcal{K}$ program will always translate into a well-formed $\mathcal{K}_\mathrm{F}$ program. This is easy to prove as most type rules for $\mathcal{K}_\mathrm{F}$ are subsumed by the $\mathcal{K}$ type rules. Thus, by proving type soundness (progress and preservation) of $\mathcal{K}_\mathrm{F}$, we show that a translated $\mathcal{K}$ program will never get stuck (modulo deadlocks, which we distinguish from unsound stuck states).

$\mathcal{K}_\mathrm{F}$ imposes no restrictions on aliasing nor does it provide any guarantees about race freedom. However, $\mathcal{K}_\mathrm{F}$ programs translated from well-formed $\mathcal{K}$ programs will always be data-race free. As $\mathcal{K}_\mathrm{F}$ itself provides no data-race guarantees, the invariants given by $\mathcal{K}$ are defined independent of well-formed configurations of $\mathcal{K}_\mathrm{F}$ (see Section 7.2 and Section 7.3).

We have a fully mechanised specification of $\mathcal{K}_\mathrm{F}$ in Coq, including a proof of type soundness, but not data-race freedom. In our hand-written proof of data-race freedom we also extend $\mathcal{K}_\mathrm{F}$ with means of tracking types and stack-boundedness of values. This has no effect on the execution or typechecking of $\mathcal{K}_\mathrm{F}$ programs and is thus excluded from the mechanised version [15]. Specifying all of $\mathcal{K}$ in Coq is future work.

7.2 Aliasing and Encapsulation

This section details the invariants on aliasing and encapsulation in well-typed $\mathcal{K}$ programs.

Safe Aliasing. One of the main technical results of this work is the proof that $\mathcal{K}$ programs enjoy *safe aliasing*, *i.e.,* two paths to the same mutable field are local to the same thread, or protected by the same lock which must be acquired before access. Informally, aliasing is safe if the following is true for all aliases x, y on the stack or heap:

- x and y have composable types, meaning they point to different parts of the same object, modulo safe `val` fields, corresponding to $\vdash t_x \otimes t_y$ in $\mathcal{K}$ (*cf.,* Figure 7).
- x and y are protected, *i.e.,* read-only aliases, or safe aliases that use locks internally, or unsafe aliases whose accesses must be wrapped in locks.
- x and y are both local to the same thread, corresponding to the `thread` capabilities of $\mathcal{K}$.
- x and y both have subordinate types, meaning that any thread accessing them must currently have exclusive access to their dominator.
- If x or y is `linear`, at least one of the aliases must be stack-bound to prevent introducing aliases of linear values on the heap.
- If one of the aliases is stack-bound, the origins of the borrowed value must be buried, to prevent multiple accessible references to the same linear value.

Proof. Part of the proof of thread-safety [15].

Strong Encapsulation of Subordinate Capabilities. Another invariant preserved by κ programs is that references in fields of subordinate type point to objects dominated by the dominator of the current enclosure. This is what grants subordinate capabilities strong encapsulation, similar to ownership types [16] and external uniqueness [17].

At run-time in κ_{F}, instances know the identity of their dominator. This identity is invariant, even under ownership transfer, because transfer operates on linear capabilities and instances of classes without a subordinate capability are their own dominators.

Let $\rightarrow$ denote "refers to" and $\iota.\texttt{dom}$ denote the dominator for an object with id ι in the heap H. Now, $\forall\, \iota, \iota' \in dom(H)$, $\iota \rightarrow \iota'$ s.t. $\iota \neq \iota'$, either one of the following holds:

1. $\iota'.\texttt{dom} = \iota.\texttt{dom}$ (a pointer between subordinates in the same enclosure)
2. $\iota'.\texttt{dom} = \iota$ (a dominator pointing to one of its subordinates)
3. $\iota.\texttt{dom} = \iota'$ (a subordinate pointing to its dominator)

or ι' is a top-level object, *i.e.,* $\iota'.\texttt{dom} = \iota'$.

Proof. Part of the proof of thread-safety [15].

7.3 Data-Race Freedom and Non-Interference.

This section describes the invariants of κ for concurrent and parallel programming.

Thread-Safety. Safe aliasing and the encapsulation guarantees mentioned above are both part of a bigger notion of a *thread-safe* (**TS**) configuration. A configuration is thread-safe if no two possible reductions can cause interference—if a possible reduction of a configuration has one thread writing to a field, there cannot be another reduction of the same configuration where the same field is read or written to by another thread.

In addition to safe aliasing, a number of constraints apply to the elements of thread-safe configurations that deal specifically with aliasing across threads:

- references in fields of subordinate type point to objects dominated by the dominator of the current enclosure;
- values of type **thread** were created by the thread that can access them;
- all local variables of **subordinate** type are dominated by the closest dominating `this` on the current stack;
- if a stack-bound `linear` value aliases a value on the heap, the value on the heap is effectively buried, *i.e.,* the only path to it is rooted on the stack;
- all accesses to values not wrapped in locks are `linear`, `thread`, `subord` or `read`.

Having defined thread-safety, we then prove that the initial configuration is **TS** and that evaluation preserves this property in a program translated from κ to κ_{F}.

▶ Preservation of Thread-Safety. In a well-formed program translated from κ, if a thread-safe configuration *cfg* can step to *cfg*′, then *cfg*′ is also thread-safe.

$$\forall\, \Gamma,\ cfg,\ cfg'.\\ \Gamma \vdash \mathbf{TS}(cfg) \wedge cfg \hookrightarrow cfg' \Rightarrow \exists \Gamma'.\Gamma' \vdash \mathbf{TS}(cfg') \wedge \Gamma \subseteq \Gamma'$$

Proof. We prove thread-safety by induction over the thread structure [15]. The proof is similar in structure to the type preservation proof of κ_{F} (and relies on type preservation to find Γ').

Data-Race Freedom. The dynamic semantics of κ_{F} tracks effect footprints in terms of reads ($\mathbf{rd}(\_)$) and writes ($\mathbf{wr}(\_)$) of object fields. This allows us to define and prove data-race freedom, which verifies that our notion of thread-safety is sound.

▶ Data-Race Freedom. If a safe configuration *cfg* steps to two different configurations causing effects Eff_1 and Eff_2 respectively, then these effects are non-conflicting:

$$\forall \Gamma,\ \mathit{cfg},\ \mathit{cfg}'\ \mathit{cfg}''\ .\\ \Gamma \vdash \mathbf{TS}(\mathit{cfg}) \wedge \mathit{cfg} \hookrightarrow^{\mathit{Eff}_1} \mathit{cfg}' \wedge \mathit{cfg} \hookrightarrow^{\mathit{Eff}_2} \mathit{cfg}'' \Rightarrow \mathit{Eff}_1 \# \mathit{Eff}_2 \vee \mathit{cfg}' = \mathit{cfg}''$$

where $\#$ denotes that two effects are disjoint or a read–read conflict:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathbf{rd}(l.f) & \# & \mathbf{rd}(l'.f') & \_(l.f) & \# & \mathbf{wr}(l'.f') \quad \text{iff} \quad l \neq l' \vee f \neq f' \\ \varepsilon & \# & \_ & \mathit{Eff}_1 & \# & \mathit{Eff}_2 \quad \text{iff} \quad \mathit{Eff}_2 \# \mathit{Eff}_1 \end{array}$$

Proof. The proof is straightforward and performed by case analysis on the thread structure, showing that any interference contradicts thread-safety [15].

Corollary: Non-Interference. Parallel κ expressions that do not use locked or unsafe capabilities are *free from interference and therefore deterministic*. The only way threads can affect each other is by writing shared mutable locations, which requires taking a lock. Without locked or unsafe capabilities there are no locks, meaning that any data that is shared between threads can only be read, and that no threads are ever blocked in their execution.

Corollary: Thread-Affinity of Thread Capabilities. Implied by **TS**, κ `thread` capabilities are thread-affine. Let $\mathit{creator}(\iota)$ return the id of the thread creating ι. From **TS** follows that in a thread *tid* with local variables V and expression e, $\forall x\,.\,V(x) = \iota \wedge\ \Gamma(x) = t \wedge\ \mathbf{thread}(t) \Rightarrow \mathit{creator}(\iota) = \mathit{tid}$ and $\forall \iota\,.\,\iota \in \mathbf{locations}(e) \wedge\ \Gamma(x) = t \wedge\ \mathbf{thread}(t) \Rightarrow \mathit{creator}(\iota) = \mathit{tid}$. Thus `thread` capabilities are only visible to their creating threads. The key elements in the type system are the $\mathbf{thread}(t) \Longrightarrow \mathbf{thread}(\mathrm{K})$ constraint in the κ rule (WF-FD) which restrict fields of type `thread` to only appear inside manifestly `thread` capabilities and the κ rule (FRAME) which does not allow `thread` capabilities to be visible in the second `async` of a `finish`.

8 Related Work

An original source of inspiration for this work was Boyland *et al.*'s "Capabilities for Sharing" which introduces a system of reference capabilities, such as immutability or ownership in a dynamic system, not amenable to static typing [10]. Similarly, κ brings together ideas from many different areas in a single system, but fully statically typed. To the best of our knowledge, the selection and integration of the features in κ are unique in an object setting:

1. Linear capabilities are similar to uniqueness [27, 34] or permissions [9, 42, 38] and enable ownership transfer [17].
2. Subordinate capabilities enable strong encapsulation similar to ownership types [16] or Universes [35, 36]. κ's combination of subordinate and dominating capabilities give arbitrary nesting, but nested aggregates may not refer to subordinate objects in their enclosing aggregate, nor does κ support incoming read-only references (owners-as-modifiers [35, 36])—both enable data-races.

3. Thread capabilities resemble thread-local heaps in Loci [43] and ownership for actors [18].
4. That linear capabilities view themselves as subordinate capabilities internally gives a form of external uniqueness [17, 26].
5. The combination of locked and read capabilities express readers–writer locks, with a compile-time guarantee that readers will not write.
6. The `safe` mode, abstracting over `read` and `locked`, avoids code duplication for traits that are agnostic to why objects are safe, similar to type qualifier generics [22, 45].
7. The combination of locked and subordinate capabilities empowers a single lock to range over an entire aggregate with a compile-time guarantee of correctness (*cf.,* owners-as-locks in *e.g.,* [16]), it also allows enforcing a crude form of lock ordering in combination with readers-writer locks by connecting lock order to nesting order (*cf.,* [7]).
8. Nested capabilities are essentially storable permissions [9, 42] but without breaking abstraction—the names of fields etc. of the object storing permissions can be kept secret.
9. The flat composition of capabilities and the packing/unpacking marries ideas from fractional permissions [9] with ownership and substructural types [13], similar to [29]. Composites of read and linear capabilities support mediation between readers and writers, using stack-bounding to identify where sharing starts and stops.

Ownership Systems. Aliasing of mutable state in object-oriented programming is a mature research field: categorisations of alias management techniques [28], ownership types [16], universe types [22, 36], external uniqueness [17], balloons [2], as well as multiple flavours of references [27, 4, 34, 3, 10, 8, 12] etc. (*cf.* [16] for a broad coverage of many aspects). Banning aliasing is usually abandoned in favour of alias control, which commonly prescribes a certain shape on the program [16, 22, 43, 18] possibly combined with an effect system to coordinate accesses to shared data across multiple program locations [19, 5, 14]. κ does not prescribe a certain topology for shared capabilities. With the shift to ubiquitous parallelism, ideas from these fields have been applied to the simplification of concurrent and parallel programming (*e.g.,* [7, 18, 26, 24, 37, 21]).

Abadi *et al.* [1] propose RaceFree Java where field declarations are associated with locks and an effect system tracks how locks are acquired and released. Classes can be parameterised with external locks. The combination of locked and subordinate capabilities in κ seem to be able to express the same, but without ghost variables or an effect system. Zhao [44] constructs a system similar to Abadi *et al.* [1] but based on fractional permissions. It uses method annotations with read/write effects and locks taken and also considers deadlocks through lock ordering, similar to [7].

The recent surge of interest in Mozilla's Rust language provides anecdotal evidence to the value of languages with data-race freedom built in. In Rust, values mediate between linear/mutable and sharable/immutable. Linear values use transfer semantics and Rust uses borrowing to simplify programming. Rust support flat unpacking of arrays, but not nested unpacking and not unpacking of other types. Rust does not have strong encapsulation meaning an aggregate's innards is not protected by the aggregate's single entry-point and no aliasing of mutable objects is allowed inside the aggregate.

Substructural Systems. κ is close in spirit to work by Caires and Seco [13] as well as work by Pottier *et al.* [38] on Mezzo, both in the context of ML-like languages. Caires and Seco formalise a fine-grained capability system for reasoning about interference caused by aliasing or concurrent accesses to aliased data with explicit synchronisation. There is no support for read-sharing or strong non-linear encapsulation.

Militão *et al.* use a substructural type system for specifying rely-guarantee protocols in a functional context [32]. Protocols capture the view of shared state from one particular alias—our capabilities are similar. Ownership transfer and recovery for linear values is supported.

The functional language Alms [40] explores the design space of practical programming with substructural types. Alms separates capabilities from references and operations that require a capability must have the capability passed in as an argument. Capabilities in Alms are lower level than in κ and can be used to express many of our capabilities. There is no unification of capabilities with building blocks like traits, or composition of capabilities.

Capability Systems and Permission Systems. Miller uses capabilities for access control and concurrency control in a distributed setting in the seminal E language [33], employing a more dynamic approach (than κ) with optional soft typing.

Mediation between different views of an object is similar to fractional permissions [9]. κ supports going from a single writer to multiple enumerable disjoint writers or readers, and in the case of readers to an unbounded number of stack-bound aliases via (E-FORWARD). Fractional permissions with nesting [11, 42] is similar to subordinate capabilities in allowing one permission to act as guard to another. These system allows turning an entire nested structure read-only. κ's subordinate capabilities are less restricted, but also not transferable. κ's `read` capabilities also provide abstraction as they allow fields remain private.

Bocchino's Deterministic Parallel Java [5] uses an effect system to guarantee deterministic parallelism for operations that have no overlapping writes. When the effect system is not enough, the user can annotate trusted operations as commuting. κ provides similar determinism guarantees when excluding locking capabilities, but resorts to locking (and non-determinism) rather than using unchecked annotations for more complex operations.

Clebsch *et al.* [20] use "deny capabilities" to provide safe sharing of objects between actors. Their capabilities always grant exclusive write access to entire objects, while κ's also allows accessing parts of an object, as well as permitting multiple parallel writers.

Westbrook *et al.* [42] formalise and implement a gradual extension to HabaneroJava, HJp, in the form of a permission system. Permissions in HJp always govern access to entire objects, and there is no notion of encapsulation modulo storing linear permission in fields which only supports tree-shaped data. When there is not enough permission information, dynamic checks are inserted which may fail, but which also allow unconstrained aliasing. Splitting a single write-permission into multiple read permissions is similar to `read` capabilities.

Chalice by Leino *et al.* [30] is a language for specification and verification of concurrent software that uses permissions to statically track aliasing. Since κ is concerned only with data-race freedom, it trades some of the more fine-grained control (*e.g.,* full method specification) for simplicity (*e.g.,* no need for manual permission tracking).

9 Discussion & Future Work

We currently require programmers to explicitly manage substructural operations manually through packing and unpacking and jails. Building simpler-to-use constructs on top of these is possible. For example, a combination of `bound` and `unpack` would remove the need for packing, at the cost of enforcing a nested structure to unpacking and packing. Employing inference to automate this to a large extent seems possible and is a direction for future work.

Implementation of κ is on-going in the actor-based language Encore. There, actors replace locks as a means of pessimistic concurrency control. Capabilities protect actors' state

while allowing ownership-transfer due to linear values. A larger case study evaluating the full expressiveness of κ is planned for future work.

Unstructured vs. Structured. We have purposely supported *unstructured* packing and unpacking of capabilities. This allows granting capabilities to other threads (possibly on other machines) without requiring the capabilities to be returned or tying their return to a particular local scope. This removes limitations inherent in effect systems (*e.g.,* [25, 19, 5]), which requires computations to be nested. Unstructured locking is important in some applications, for example to implement hand-over-hand locking, and is a possible direction for future work.

Revocation. We only consider "cooperative revocation", *i.e.,* there is no built-in mechanism to arbitrarily revoke a given capability. In the security setting from which the capability idea stems, this is a major concern but it makes less sense in our setting as revoking a capability from another thread at an unfortunate point in time might cause system-wide inconsistencies.

Other Capabilities. The only form of dynamic concurrency control considered in this paper is locks. In ongoing work, the set of modes are extended with `async` (objects are actors), `atomic` (objects use transactional memory) and `lockfree` (lock-free programming). In this richer setting, we aim to address more programmer-friendly forms of multi-object atomicity.

10 Conclusions

Creating and destroying aliases enables and constrains parallelism and is key to establishing data-race freedom and non-interference. By capturing how data is shared and accessed through modes, and by introducing a structured approach to creating and destroying aliases through the combination of capabilities that make up classes, data-race freedom can be guaranteed statically with or without dynamic concurrency control. κ's invariants are similar to what an effect system can give, but avoids complicated effect annotations which propagate through the program and constrain inheritance. Tracking modes in types provides machine-checked documentation about alias freedom and sharing which localises reasoning.

The unification of traits and capabilities allows a single trait to serve multiple concurrency scenarios, which extends trait-based reuse. It also simplifies programming as trait implementers may safely assume data-race freedom. Ultimately, κ brings together a broad spectrum of prior work in a unified system.

Acknowledgements. We are grateful for the comments from Sophia Drossopoulou, the SLURP reading group at Imperial College, Dave Clarke and the anonymous reviewers.

References

1 Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, March 2006. doi:10.1145/1119479.1119480.

2 Paulo Sérgio Almeida. *Control of Object Sharing in Programming Languages*. PhD thesis, Imperial College London, June 1998.

3 David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a Dialect of Java Without Data Races. In *OOPSLA*, pages 382–400, 2000.

4 Henry G. Baker. 'Use-once' Variables and Linear Objects – Storage Management, Reflection and Multi-Threading. *ACM SIGPLAN Notices*, 30(1), January 1995.

5 Robert Bocchino. An Effect System and Language for Deterministic-By-Default Parallel Programming, 2010. PhD thesis, University of Illinois at Urbana-Champaign.

6 Shekhar Borkar and Andrew A Chien. The future of microprocessors. *CACM*, 54(5):67–77, 2011.

7 Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.

8 John Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, May 2001.

9 John Boyland. Checking interference with fractional permissions. In *SAS*, pages 55–72, 2003.

10 John Boyland, James Noble, and William Retert. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. ECOOP. Springer, 2001.

11 John Tang Boyland. Semantics of fractional permissions with nesting. *ACM Trans. Program. Lang. Syst.*, 32(6):22:1–22:33, August 2010. doi:10.1145/1749608.1749611.

12 John Tang Boyland and William Retert. Connecting Effects and Uniqueness with Adoption. In *POPL*, pages 283–295, 2005.

13 Luís Caires and João C. Seco. The Type Discipline of Behavioral Separation. POPL, 2013.

14 Nicholas R. Cameron, Sophia Drossopoulou, James Noble, and Matthew J. Smith. Multiple ownership. In *OOPSLA*, 2007.

15 E. Castegren and T. Wrigstad. Reference capabilities for trait based reuse and concurrency control. Technical Report 2016-007, 2016. Uppsala University.

16 Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming*, volume 7850 of *LNCS*. Springer, 2013.

17 Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP*, 2003.

18 Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Johnsen. Minimal ownership for active objects. In *Programming Languages and Systems*, volume 5356 of *LNCS*. Springer, 2008.

19 David G. Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *OOPSLA*, pages 292–310, 2002.

20 Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *AGERE*, 2015.

21 David Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universe Types for Race Safety. In *VAMP*, 2007. URL: http://pubs.doc.ic.ac.uk/universes-races/.

22 Werner M. Dietl. *Universe Types: Topology, Encapsulation, Genericity, and Tools*. Ph.D., Department of Computer Science, ETH Zurich, December 2009.

23 M. Fähndrich and R. DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *PLDI*, pages 13–24, 2002.

24 Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and Reference Immutability for Safe Parallelism. In *OOPSLA*, pages 21–40, 2012.

25 Aaron Greenhouse and John Boyland. An object-oriented effects system. ECOOP, 1999.

26 Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *ECOOP*, 2010.

27 John Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *OOPSLA*, 1991.

28 John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard Holt. The Geneva Convention on the Treatment of Object Aliasing. *OOPS Messenger*, 3(2), April 1992.

29 Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. Superficially Substructural Types. ICFP, New York, NY, USA, 2012. ACM.

30 K Rustan M Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*. 2009.

31 H. Levy, editor. *Capability Based Computer Systems*. Digital Press, 1984.

32 Filipe Militão, Jonathan Aldrich, and Luís Caires. Rely-guarantee protocols. In *ECOOP*, 2014.

33 Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

34 Naftaly Minsky. Towards alias-free pointers. In *ECOOP*, July 1996.

35 P. Müller and A. Poetzsch-Heffter. Universes: a type system for controlling representation exposure. Technical Report 263, 1999. Fernuniversität Hagen.

36 Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer, 2002.

37 Pratibha Permandla, Michael Roberson, and Chandrasekhar Boyapati. A type System for Preventing Data Races and Deadlocks in the Java Virtual Machine Language. In *LCTES*, pages 1–10, 2007.

38 Francois Pottier and Jonathan Protzenko. Programming with Permissions in Mezzo. In *ICFP*, pages 173–184, September 2013.

39 Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and AndrewP. Black. Traits: Composable units of behaviour. In *ECOOP*, 2003.

40 Jesse A. Tov. *Practical Programming with Substructural Types*. PhD thesis, Northeastern University, 2012.

41 Philip Wadler. Linear Types Can Change the World! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*. North Holland, 1990.

42 Edwin Westbrook, Jisheng Zhao, Zoran Budimli, and Vivek Sarkar. Practical permissions for race-free parallelism. In *ECOOP*, 2012.

43 Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. Loci: Simple thread-locality for Java. In *ECOOP*, 2009.

44 Yang Zhao. *Concurrency Analysis Based On Fractional Permission System*. PhD thesis, University of Wisconsin – Milwaukee, 2007.

45 Yoav Zibin, Alex Potanin, Shay Artzi, et al. Object and reference immutability using Java generics. In *ESEC/FSE*. 2007.

A Calculus for Variational Programming*

Sheng Chen[1], Martin Erwig[2], and Eric Walkingshaw[3]

1 **University of Louisiana at Lafayette**
`chen@louisiana.edu`
2 **Oregon State University**
`erwig@oregonstate.edu`
3 **Oregon State University**
`walkiner@oregonstate.edu`

Abstract

Variation is ubiquitous in software. Many applications can benefit from making this variation explicit, then manipulating and computing with it directly—a technique we call "variational programming". This idea has been independently discovered in several application domains, such as efficiently analyzing and verifying software product lines, combining bounded and symbolic model-checking, and computing with alternative privacy profiles. Although these domains share similar core problems, and there are also many similarities in the solutions, there is no dedicated programming language support for variational programming. This makes the various implementations tedious, prone to errors, hard to maintain and reuse, and difficult to compare.

In this paper we present a calculus that forms the basis of a programming language with explicit support for representing, manipulating, and computing with variation in programs and data. We illustrate how such a language can simplify the implementation of variational programming tasks. We present the syntax and semantics of the core calculus, a sound type system, and a type inference algorithm that produces principal types.

1998 ACM Subject Classification F.3.3 Logics and Meanings of Programs, D.3.2 Programming Languages

Keywords and phrases Variational programming, variational types, variability-aware analyses

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.6

1 Introduction

The idea of representing and computing with *explicit variation* has been used to solve a variety of problems where a similar computation must be performed on sets of related data. For example, a challenge in *software product lines* (SPL) is to ensure that *all* program variants that can be generated from a code base satisfy a given property. Since analyzing each variant in sequence is intractable, researchers have developed algorithms and tools for analyzing the variational code directly. This work enables the efficient parsing [29], type checking [28, 46, 2], type inference [17, 16], model checking [19, 1], and flow analysis [6, 7] of whole software product lines at once.

But the need to write programs that manipulate variation representations is not limited to the area of software product lines. Other applications include computing with alternative privacy policies [3], improving type error messages [10, 11, 15], improving the performance

* This work is supported by the National Science Foundation under the grants CCF-1219165 and IIS-1314384.

of simulations [45], supporting view-based editing [53], and several applications to software testing [30, 31, 39, 49].

All these applications require an explicit representation of variation in data along with operations for inspecting, transforming, and/or mapping computations over such variational data. We collectively refer to these as *variational programming* tasks. Variational programming is not well supported by existing programming languages. While there exists some support from libraries [24, 29], there is no dedicated language support for variational programming. The lack of language support makes it difficult to reason about the variational programs and the artifacts they produce.

Therefore, we present in this paper a set of carefully designed, dedicated variational programming abstractions that can provide guarantees about variational programs and their results. The integration of these operations into a functional programming language can greatly improve the effectiveness, reliability, and productivity of variational programming.

In the remainder of this paper, we first motivate variational programming and illustrate it with several examples. We then formalize the underlying principles by defining a calculus for variational programming. This paper makes the following contributions.

- We explain the idea of variational programming through several examples in Section 2. We introduce the underlying variation representation and demonstrate how variational programming combinators can be built from a small set of core constructs and principles.
- We present the syntax and operational semantics of a variational programming calculus (VPC) in Section 3. A particular challenge is to balance and integrate the competing notions of variation-preserving computation and variation aggregation. A careful choice of congruence rules and evaluation strategy ensures the confluence of the semantics.
- We develop a type system for VPC in Section 4 and present a soundness theorem. Named choices lead to a limited form of dependent types, called *dimension polymorphism*, which reflect variability in expressions at the type level. Moreover, since operations for the elimination of variation cause subtle changes in variational types, we have to employ constraints in the type system to track the evolution of variability in computations.
- In Sections 5 and 6, we present constraint generation and constraint solving that constitute a type inference algorithm for VPC, which is sound, complete, and principal.

We discuss related work in Section 7 and conclude the paper with Section 8.

2 Variational Programming

The term "variational program" has two related meanings. On the one hand, it refers to a program family from which individual programs can be selected. On the other hand, it refers to programs that process variational values. For variational programs that process variational programs the two meanings collapse. In this paper we are primarily focused on the second meaning, that is, we want to study programs that process variational values.

The systematic processing of variational values requires machinery to create, query, and transform such values. These tasks should be supported by a principled variation representation that enjoys a rich set of laws. In Section 2.1 we present a simple representation based on the choice calculus [23], illustrate how it can be embedded into a (functional) programming language, and then motivate and outline the rest of this section.

2.1 Representing Variation

We represent variation by named, binary choices [23]. For example, a program that applies a function `f` to either `1` or `True` can be represented as `p = \f.f A<1,True>` where the choice represents a point of variation within `p`. A program that contains choices is called a *variational program*. This variational program encodes two plain programs, `\f.f 1` and `\f.f True`, called *variants*. These variants can be obtained by *selecting* either the left or right alternative from the choice. The name associated with the choice, A, is called a *dimension*. Dimensions allow the synchronization of choices in different parts of a program. For example, the program `q = p A<succ,not>` runs `p` with a choice of functions that will be synchronized with the choice of constants. Variational programs can vary in more than one dimension. Choices in different dimensions vary independently of one another, that is, whereas `q` contains only two variants, the variational program `q' = p B<succ,not>` contains four.

Executing a variational program produces a variational result. For example, executing program `q` above produces the result `A<2,False>`. This result may, in general, vary in all of the dimensions referred to in the variational program. We call such executions *variation-preserving* (see Section 2.2).

However, variational programming encompasses more than just executing variational programs. For example, suppose we have implemented a variational function that computes the time needed to complete a project depending on many design decision each represented by a different dimension. We can imagine several subsequent operations to perform on the variational result of this function. For example, we may want to do further variation-preserving computations by adding the variational time to complete the project to the variational time needed to complete another project, or by comparing it with a variational time computed by a different function. We may also want to *filter* the result by making selections that reduce the amount of variability in the result, which can help speed up subsequent variation-preserving computations. Alternatively, we may want to *aggregate* the variation, for example, to identify the minimum or maximum variant, to determine how to proceed depending on if we are being paid on salary or hourly. In the rest of this section, we take a closer look at variation-preserving computations (Section 2.2), reducing variation through selection (Section 2.3), and variation aggregation (Section 2.4), introducing the core constructs of our language and deriving some general combinators to support variational programming, as illustrated through an example application (Section 2.5). Finally, in Section 2.6, we argue why our language design is a good basis for variational programming.

2.2 Variation-Preserving Computations

We start with the task of adding variational numbers. Since choices in the same dimension are synchronized, expect evaluating `A<1,2> + A<3,4>` to produce `A<4,6>`. But how about `A<1,2> + B<3,4>` or even `A<1,2> + 3`? We can address the latter expression by realizing that a non-variational number is the same as a choice whose alternatives are identical, that is, `3 = A<3,3>`. In other words, choices are *idempotent*. Therefore, `A<1,2> + 3 = A<1,2> + A<3,3> = A<4,5>`. We arrive at the same result by considering that the variational expression `A<1,2> + 3` represents two plain expressions `1+3` and `2+3`, depending on which alternative from dimension `A` is selected, that is, the expression is equivalent to `A<1+3,2+3>`, which evaluates to `A<4,5>`.

We can see that the operation `+3` was effectively "moved" into the `A` choice, which is an application of the *choice distribution* law [23] that allows one to push a common context into a choice. This is illustrated below where C is an arbitrary context and D is an arbitrary dimension.

$$C[D\langle e, e'\rangle] = D\langle C[e], C[e']\rangle$$

Choice distribution is the basis for variation-preserving computation. It says that we can evaluate an expression such as `f A<3,4>` by applying `f` to the alternatives of the `A` choice. With this we can also determine the result of `A<1,2> + B<3,4>` by moving one of the choices into the other and then doing this again in each of the alternatives.

```
A<1,2>+B<3,4> = A<1+B<3,4>,2+B<3,4>> = A<B<1+3,1+4>,B<2+3,2+4>> = A<B<4,5>,B<5,6>>
```

If we decide to move the `A` choice instead, we obtain the equivalent value `B<A<4,5>,A<5,6>>`. This can be verified by making all possible selections for `A` and `B`.

Previously, we argued that `A<1,2> + A<3,4>` yields only two variants since both `A` choices are synchronized. It is instructive to perform choice distribution on this example to illustrate that selection commutes with semantics-preserving computation. In fact, we obtain the same derivation except that all `B`s are replaced by `A`s, so the resulting expression is `A<A<4,5>,A<5,6>>`. Observe that both `5` values can never be selected since selecting either the left or right alternative of `A` implies the same selection in the nested `A` choices. This phenomenon, called *choice domination*, neutralizes the effect of nested choices in the same dimension on the variability of values. These examples illustrate that we can apply a (variational) function to a (variational) value, and choice distribution will produce a result in which the variation in both the function and argument are preserved.

As another example, we can also compare variational values by a variation-preserving computation. For example, the expression `A<3,5> == A<4,5>` evaluates to `A<False,True>`, and `A<B<5,4>,7>` ≤ `A<6,C<8,2>` yields `A<True,C<True,False>>`.

2.3 Eliminating Variation

The elimination of choices is called *selection*. Selection takes a selector of the form $D.s$ (where D is a dimension and s is either `L` or `R`), traverses the expression, and replaces all choices named D with its corresponding left or right alternative. For example, given program `q` defined in Section 2.1, the expression `sel A.L q` produces the variant `(\f.f 1) succ`. Since the program `q'` contains choices in two different dimensions, we need two selection operations to eliminate the variability in the program and obtain plain expressions. By considering all four possible selections on `q'`, we can observe that only two of the variants are type correct.

```
sel A.L (sel B.L q') = (\f.f 1) succ      sel A.R (sel B.L q') = (\f.f True) succ
sel A.L (sel B.R q') = (\f.f 1) not       sel A.R (sel B.R q') = (\f.f True) not
```

A choice is an expression, not a textual macro.[1] This means that we cannot vary, say, a few letters in the middle of an identifier or optionally exclude a closing parenthesis. However, we can pass choices to functions, and observe and manipulate them at runtime.

2.4 Variation Aggregation

Besides selecting individual variants, we often want to systematically eliminate all the variability in a variational program. For example, to compute the minimum variant of a variational number, we have to eliminate all of the choices, replacing each one by the smaller of its two alternatives. This kind of aggregation is supported by the syntactic form **any** d **from** e **in** e' **else** e''. An **any**-expression checks to see if there are any choices in

[1] This is in contrast with, for example, the C Preprocessor's `#ifdef` notation, which is widely used to implement static variation in C programs.

expression e. If so, it picks one out, binds the corresponding dimension name to a new variable d, then returns expression e', which may contain references to d. If no choices exist in e, the **any**-expression evaluates to e''. The following function `vmin` uses **any** to implement the minimum-variant aggregation. If a choice is found in its argument `i`, it eliminates the choice with `sel` and recursively computes the minimum variant of the left and right alternatives, then returns the minimum of those two values (using the function `min`). If no choice is found, then all variation has been eliminated and `i` is a plain integer, which is simply returned.

```
vmin : Int -> Int
vmin i = any d from i in
             min (vmin (sel d.L i)) (vmin (sel d.R i))
         else i
```

Now consider the expression `vmin A<B<4,3>,2>`, which we expect to evaluate to `2`. However, by applying the choice distribution principle from the previous subsection, the expression could also evaluate to `A<B<4,3>,2>`. In other words, in this scenario, choice distribution prevents aggregation and changes the intended semantics of the program. There are several ways to address this issue, but we choose to disambiguate the situation syntactically, which leads to a simpler operational semantics. Specifically, we allow certain function arguments to be annotated as "aggregating" (using `@`), which acts as a boundary that prevents choice distribution. In the case of `vmin` this looks as follows.

```
vmin @i = any d from i in ...
```

Now `vmin A<B<4,3>,2>` will compute `2` and not return a choice. But what if we want to independently compute `vmin` for two alternatives of a dimension, say `A`? We can still do this by applying a corresponding choice of `vmin`s, that is, we write `A<vmin,vmin> A<B<4,3>,2>`. Now choice distribution will preserve the `A` dimension and we get the result `A<3,2>`. However, note that the idempotency law does not hold for variation aggregators, that is, in general we have `A<vmin,vmin>` $\neq$ `vmin`.

Within the definition of `vmin` are two inter-related patterns that occur frequently in variational programs. The first is to select both the left and right alternatives of the same dimension in parallel; the second is to pick an arbitrary dimension, then immediately select with it. To accommodate the first pattern, we introduce a derived form, **split**, which allows us to rewrite `vmin` as follows.

```
vmin @i = any d from i in
            split i on d<l,r> -> min (vmin l) (vmin r)
          else i
```

In the **split** expression, `d` refers to the dimension we want to eliminate, bound by the enclosing **any** expression; `l` and `r` are new names bound to the result `sel d.L i` and `sel d.R i`, respectively.

To accommodate the second pattern, we combine **split** and **any** into a single construct as illustrated below. This expression form picks a dimension out of `i` and immediately splits on it.

```
vmin @i = split i on any
                d<l,r> -> min (vmin l) (vmin r)
           else i
```

Which dimension is picked by **any** is determined by a fixed, predefined ordering among dimensions. Therefore the names of dimensions in a variational value generally matters. However, in functions that systematically process all variability present in a value, the order does not matter. For example, `vmin A<B<5,4>,7>` will compute the same result, independent of the order in which the dimensions are picked by **any**. To demonstrate, here is the computation that unfolds if `A` is picked first.

```
min (vmin B<5,4>) (vmin 7) = min (min 5 4) 7 = min 4 7 = 4
```

If `B` is picked first, the enclosing `A` choice is preserved across both alternatives of `B`.

```
min (vmin A<5,7>) (vmin A<4,7>) = min (min 5 7) (min 4 7) = min 5 4 = 4
```

The two computations return the same result because choices commute [23] allowing arbitrary reordering of how choices are nested. Specifically, for any two dimensions, `A` and `B`, the following relationship holds.

```
A<B<a,b>,B<c,d>>  ≡  B<A<a,c>,A<b,d>>
```

Computing the minimum variant of a variational integer is an instance of the more general task of aggregating variability, which can be captured by a variational join function that accumulates variants with a binary function.

```
vjoin : (a -> a -> a) -> a -> a
vjoin f @x = split x on any
                   d<l,r> -> f (vjoin f l) (vjoin f r)
               else x
```

Using `vjoin` we can define `vmin` more simply as `vjoin min`. More generally, we can define a variational fold that both converts each variant to a separate accumulator type `b`, then aggregates them.

```
vfold : (a -> b) -> (b -> b -> b) -> a -> b
vfold f g @x = split x on any
                     d<l,r> -> g (vfold f g l) (vfold f g r)
                 else f x
```

Now `vjoin` is just a `vfold` where the conversion is the identity function: `vjoin = vfold id`.

Using `vfold`, we can aggregate variational values in a variety of ways. For example, here is a function for counting the number of choices in a variational value.

```
choices : a -> Int
choices = vfold (\x.0) (\x y.1+x+y)
```

The `vfold` function is essentially a bottom-up tree fold over the binary trees of choices. However, because of the semantics of **any**, the aggregation order is determined not by the structure of the value, but by the ordering of dimension names contained in the value. This means that one should be careful when using `vfold` or `vjoin` with aggregating functions that are not associative.

2.5 An Application to Variational Unification

Variational unification finds substitutions that solve equations of the form: $A\langle \texttt{Int}, a\rangle \equiv^{?} B\langle b, c\rangle$. This is not a trivial problem. The following substitution is an obvious solution to this example.

$\sigma_1 = \{a \mapsto \texttt{Int},\ b \mapsto \texttt{Int},\ c \mapsto \texttt{Int}\}$

However, while some simple unifiers such as σ_1 can be found quickly, identifying the most general unifier is not easy. To wit, here is a list of some other solutions, some more general than others, some unrelated. Substitution σ_6 is the most general unifier.

$$\sigma_2 = \{b \mapsto A\langle \texttt{Int}, a\rangle, c \mapsto A\langle \texttt{Int}, a\rangle\} \qquad \sigma_3 = \{a \mapsto B\langle \texttt{Int}, f\rangle, b \mapsto \texttt{Int}, c \mapsto A\langle \texttt{Int}, f\rangle\}$$
$$\sigma_4 = \{a \mapsto B\langle f, \texttt{Int}\rangle, b \mapsto A\langle \texttt{Int}, f\rangle, c \mapsto \texttt{Int}\} \qquad \sigma_5 = \{a \mapsto B\langle d, f\rangle, b \mapsto A\langle \texttt{Int}, d\rangle, c \mapsto A\langle \texttt{Int}, f\rangle\}$$
$$\sigma_6 = \{a \mapsto B\langle A\langle i, d\rangle, A\langle j, f\rangle\rangle, b \mapsto B\langle A\langle \texttt{Int}, d\rangle, g\rangle, c \mapsto B\langle h, A\langle \texttt{Int}, f\rangle\rangle\}$$

One particular challenge for the unification algorithm is to find substitutions modulo an equivalence relation ($\equiv$) that includes laws for choice distribution, idempotency, and domination, which have been described already, plus some others.

A variational unification algorithm was first presented in [17]. This algorithm has since been employed as a crucial component in a number of applications [17, 16, 10, 12, 13, 14]. The original implementation in Haskell required a considerable number of auxiliary type and function definitions to support handling choice types. Having choices available as part of the language simplifies the implementation tremendously, allowing the programmer to focus on the main logic. First, the data type definition for types does not need to mention choice types and only needs to provide the (non-variational) constructors to represent the original (non-variational) core type language.

```
data Type = TInt | TBool | TVar Int | TFun Type Type
```

The unification of choice types works by systematically matching dimensions and type constructors, constructing substitutions along the way. The matching process is dominated by the matching of dimensions since they can be moved up or down in expressions (due to choice distribution/factoring), which is directly supported by the operation `split`. This is reflected in the following definition of the function `vunify`. Without going into too much detail, we explain how some of the cases work, and in particular, how they can profitably exploit the fact that variation is built into the language.

```
vunify : Type -> Type -> Subst
vunify @t @u = split t on any d<lt,rt> ->
                 split u on d<lu,ru> -> d<vunify lt lu,vunify rt ru>        (1)
                 else                   d<vunify lt  u,vunify rt  u>        (2)
               else
                 split u on any d<lu,ru> -> d<vunify t lu,vunify t ru>      (3)
                 else                       unify t u                       (4)
```

In case (1), if both types contain the same dimension, `vunify` computes a choice of substitutions for both alternatives, which is equivalent to a substitution with choice types.

$$D\langle\{a \mapsto T, \ldots\}, \{a \mapsto T', \ldots\}\rangle = \{a \mapsto D\langle T, T'\rangle, \ldots\}$$

Since it can happen that either substitution contains a mapping for a variable that isn't contained in the other, choice factoring can be "blocked" in such cases. We can write a function that explicitly converts a variational substitution into a substitution containing choice types and that takes care of these cases by adding a type variable for missing substitutions; we omit the code here for brevity.

In case (2), if `t` contains some dimension `d` but `u` does not, we can unify each of `t`s alternatives with `u` due to idempotency (that is, `u` = `D<u,u>`). The case (3) is symmetric. Finally, in case (4), neither type contains any more dimensions, so we invoke the non-variational function `unify`, which implements Robinson's traditional unification algorithm.

This example illustrates how variational concerns can be isolated from the rest of the unification algorithm; `vunify` focuses on variation, while `unify` handles the non-variational type structure.

2.6 A Foundational Language for Variational Programming

We argue that the features described in this section constitute a rational basis for a variational programming language. A choice between two alternatives is by definition the simplest possible representation of variation, and the choice calculus has a well-developed theory [23, 50] that has been successfully reused in several contexts [17, 16, 10, 12, 13, 14, 24, 52, 53, 11, 15]. Choices are also used internally in the TypeChef system [29], which has been reused in a number of projects for analyzing software product lines [37, 38, 33, 30, 34], and choices are the basic representation used in other variational programming scenarios outside of software product lines [3, 48]. Choices encode variation *in-place* but can also be combined with compositional approaches to variation [51]. Choices between several alternatives can be modeled by cascading choices in different dimensions. In previous work [53], we have shown how the choice calculus can be extended with more general conditions on choices, but we omit this extension here to keep the presentation concise.

The concept of variation-preserving computation describes the property that selection commutes with evaluation. That is, running a variational program corresponds to running all of the individual variants separately. This is the philosophy expressed in almost all work on analyzing software product lines [47] and in recent work on variational execution [39, 3]. Selection is a symmetric elimination form for choices that satisfies Gentzen's principle [25, p. 102]. This means that choice and selection are information-preserving inverses, a property that enables simple and reversible semantics.

Finally, a necessary feature for practical programming with variations is *reflection* on variability. In Section 2.4, we describe the **any** form for extracting one arbitrary dimension if it exists, and in the next section we describe a similar **the** form that extracts one specific dimension if it exists. The **any** and **the** forms are orthogonal constructs that are sufficient to implement pattern matching on variations, as illustrated by the derived form **split**.

3 Syntax and Semantics of VPC

In this section we define a variational programming calculus (VPC). The calculus extends the lambda calculus with all of the features needed to implement the variational functional programming language described in the previous section. We define the syntax of VPC in Section 3.1 and a small-step operational semantics in Section 3.2.

3.1 Syntax

We separate the definition of VPC into two parts: (1) a core calculus that includes only the essential features of the language, and (2) several derived forms, defined in terms of the core calculus, that provide convenient surface syntax for variational programming.

Core calculus.

The syntax of the core calculus is defined in Figure 1. The first two lines define separate namespaces for variables that refer to expressions and variables that refer to dimension names. In this paper, we distinguish these namespaces by using letters from the beginning of the alphabet for dimension variables, and from the end of alphabet for expression variables.

$$
\begin{array}{rcll}
v & ::= & x \mid y \mid z \mid \ldots & (\textit{expression variables}) \\
d & ::= & a \mid b \mid c \mid \ldots & (\textit{dimension variables}) \\
D & ::= & A \mid B \mid C \mid \ldots & (\textit{dimension names}) \\
\delta & ::= & d \mid D & (\textit{dimensions}) \\
\varphi & ::= & v \mid d & (\textit{variables}) \\
s & ::= & \ell \mid r & (\textit{selectors}) \\
* & ::= & @ \mid \epsilon & (\textit{abstraction annotations}) \\
\\
e & ::= & \kappa \mid \lambda^{*}\varphi.e \mid e\ e \mid v & (\textit{lambda calculus + constants}) \\
 & \mid & \textbf{let}\ v = e\ \textbf{in}\ e & (\textit{recursive let}) \\
 & \mid & \delta & (\textit{dimensions}) \\
 & \mid & \delta\langle e, e\rangle & (\textit{choice}) \\
 & \mid & \delta_s \blacktriangleright e & (\textit{selection}) \\
 & \mid & \textbf{any}\ d\ \textbf{from}\ e\ \textbf{in}\ e\ \textbf{else}\ e & (\textit{any dimension pick}) \\
 & \mid & \textbf{the}\ \delta\ \textbf{from}\ e\ \textbf{in}\ e\ \textbf{else}\ e & (\textit{specific dimension pick}) \\
\\
\underline{e} & ::= & \kappa \mid D \mid \lambda^{*}\varphi.e \mid D\langle \underline{e}, \underline{e}\rangle & (\textit{values})
\end{array}
$$

Figure 1 Syntax of core VPC.

Literal dimension names are represented by capital letters. The syntactic category δ includes both dimension variables and dimension names, while φ includes both dimension variables and expression variables. As a convention, we use the nonterminal of a syntactic category (or variations on it) to stand for arbitrary instances of that category. For example, φ, φ_1, and φ_2 all generically refer to variables where it is not important to distinguish between expression variables and dimension variables. We explain the syntactic categories s and $*$ in the context of the relevant expression forms.

The metavariable e ranges over VPC expressions. The first two lines enumerate the constructs of the lambda calculus extended by constants, ranged over by κ, and recursive **let**-expressions. Abstractions differ from standard lambda calculus in two ways. First, they can be distinguished by the namespace of their declared variable. That is, we will sometimes distinguish between expression abstractions, $\lambda v.e$, and dimension abstractions, $\lambda d.e$. Second, they may be optionally annotated by an @ symbol, as in $\lambda^{@}\varphi.e$, which prevents mapping the function over variation in its argument, as described in Section 2.4. Note that the alternative annotation ϵ represents the *lack* of an @ annotation; that is, we write simply $\lambda\varphi.e$ for an unannotated abstraction. In addition to constants and expression variables, the third line defines that we can also refer to dimension literals and dimension variables.

The next two lines define the choice and select constructs, described in Section 2.1 and Section 2.3, for introducing and eliminating variation, respectively. The dimension δ associated with a choice $\delta\langle e_1, e_2\rangle$ may be either a literal dimension name D or a reference to a dimension variable d. A selection replaces, within e, all choices in dimension δ by either their left alternatives, written $\delta_\ell \blacktriangleright e$, or their right alternatives, written $\delta_r \blacktriangleright e$.

The **any** d **from** e **in** e' **else** e'' and **the** δ **from** e **in** e' **else** e'' forms match against the variability in e. The **any** form is described in Section 2.4, while **the** is new.

For **any**, if e contains variation, then d will be bound to a dimension from e and e' will be evaluated, otherwise d will remain undefined and e'' will be evaluated. For **the**, if e contains variation in the specified dimension δ, then e' will be evaluated, otherwise e'' will

$$
\begin{array}{ll}
\textbf{split}\ e\ \textbf{on}\ \delta\langle v_\ell, v_r\rangle \to e' & \rightsquigarrow (\lambda v_\ell\, v_r.e')\ (\delta_\ell \blacktriangleright e)\ (\delta_r \blacktriangleright e) \\
\textbf{split}\ e\ \textbf{on}\ \delta\langle v_\ell, v_r\rangle \to e'\ \textbf{else}\ e'' & \rightsquigarrow \textbf{the}\ \delta\ \textbf{from}\ e\ \textbf{in split}\ e\ \textbf{on}\ \delta\langle v_\ell, v_r\rangle \to e'\ \textbf{else}\ e'' \\
\textbf{split}\ e\ \textbf{on any}\ d\langle v_\ell, v_r\rangle \to e'\ \textbf{else}\ e'' & \rightsquigarrow \textbf{any}\ d\ \textbf{from}\ e\ \textbf{in split}\ e\ \textbf{on}\ d\langle v_\ell, v_r\rangle \to e'\ \textbf{else}\ e'' \\
\textbf{ifvar}\ e\ \textbf{then}\ e'\ \textbf{else}\ e'' & \rightsquigarrow \textbf{any}\ d\ \textbf{from}\ e\ \textbf{in}\ e'\ \textbf{else}\ e'' \\
\textbf{ifplain}\ e\ \textbf{then}\ e'\ \textbf{else}\ e'' & \rightsquigarrow \textbf{any}\ d\ \textbf{from}\ e\ \textbf{in}\ e''\ \textbf{else}\ e'
\end{array}
$$

Figure 2 Derived syntactic forms for VPC.

be evaluated. Note that **any** *declares* a new dimension variable d while **the** *refers* to an existing dimension name or variable δ.

Values.

The last line in Figure 1 describes the *values* of VPC as a subset of expressions. Values are produced by fully evaluating well-typed and terminating VPC expressions. In addition to constants and abstractions, which are typical values in lambda calculus, a value can also be dimension literal or a choice whose alternatives are also values. Note that only choices with literal dimension names may be values. This implies that expressions with free dimension variables are not well-typed (see Section 4). Additionally, observe that the body of an abstraction value may be an arbitrary expression.

Derived forms.

Figure 2 extends the syntax of VPC with several derived forms that macro-expand to constructs in the core calculus. These forms enable higher-level operations on variational values without complicating the semantics and type system of VPC.

The first group of derived forms are three variations on the **split** construct described in Section 2.4. A basic **split** expression makes both possible selections on e in dimension δ and binds these to v_ℓ and v_r in e' using function application. The **split**–**else** form expands to a **the** expression that checks whether δ appears in a free choice in e; if so, it performs the split, otherwise it returns e''. Finally, the **split**–**any**–**else** form expands to an **any** expression that picks an arbitrary dimension out of e, if one exists, then either splits on that dimension or returns e'' if there is no variation in e.

The second group introduces two derived forms for basic queries about the presence (**ifvar**) or non-presence (**ifplain**) of variability in e. These expand to **any**-expressions that declare an arbitrary fresh dimension variable d, which is not referenced in e'.

3.2 Operational Semantics

In this section we define a small-step operational semantics for VPC. Since derived forms macro-expand into the core calculus before evaluation, we consider only the core calculus here. The step relation has the form $e \longrightarrow e'$ and is defined in Figure 3. We separate the discussion of the rules into three parts: (1) reduction rules, which form the core of the semantics, (2) commutation rules for setting up future reductions, and (3) congruence rules for enabling reduction in subexpressions.

CHC-ELIM-L
$D_\ell \blacktriangleright D\langle e_1, e_2\rangle \longrightarrow D_\ell \blacktriangleright e_1$

CHC-ELIM-R
$D_r \blacktriangleright D\langle e_1, e_2\rangle \longrightarrow D_\ell \blacktriangleright e_2$

SEL-IDEMP-K
$\delta_s \blacktriangleright \kappa \longrightarrow \kappa$

SEL-IDEMP-D
$\delta_s \blacktriangleright D \longrightarrow D$

APP-REDUCE
$(\lambda\varphi.e)\ e' \longrightarrow \textbf{split}\ e'\ \textbf{on any}\ d\langle v_\ell, v_r\rangle \to d\langle(\lambda\varphi.e)\ v_\ell, (\lambda\varphi.e)\ v_r\rangle\ \textbf{else}\ [e'/\varphi]e$

APP$^@$-REDUCE
$(\lambda^@\varphi.e)\ e' \longrightarrow [e'/\varphi]e$

ANY-REDUCE-THEN
$\textbf{any}\ d\ \textbf{from}\ \underline{e}\ \textbf{in}\ e'\ \textbf{else}\ e'' \longrightarrow [\min(\mathit{dims}(\underline{e}))/d]e'$

ANY-REDUCE-ELSE
$\varnothing = \mathit{dims}(\underline{e}) \Rightarrow \textbf{any}\ d\ \textbf{from}\ \underline{e}\ \textbf{in}\ e'\ \textbf{else}\ e'' \longrightarrow e''$

THE-REDUCE-THEN
$\delta \in \mathit{dims}(\underline{e}) \Rightarrow \textbf{the}\ \delta\ \textbf{from}\ \underline{e}\ \textbf{in}\ e'\ \textbf{else}\ e'' \longrightarrow e'$

THE-REDUCE-ELSE
$\delta \notin \mathit{dims}(\underline{e}) \Rightarrow \textbf{the}\ \delta\ \textbf{from}\ \underline{e}\ \textbf{in}\ e'\ \textbf{else}\ e'' \longrightarrow e''$

LET-REDUCE
$\textbf{let}\ v = e\ \textbf{in}\ e' \longrightarrow [[\textbf{let}\ v' = e\ \textbf{in}\ v'/v]e/v]e'$

SEL-LET
$\delta_s \blacktriangleright \textbf{let}\ v = e\ \textbf{in}\ e' \longrightarrow \textbf{let}\ v = \delta_s \blacktriangleright e\ \textbf{in}\ \delta_s \blacktriangleright e'$

APP-CHC
$\delta\langle e_1, e_2\rangle\ e' \longrightarrow \delta\langle e_1\ (\delta_\ell \blacktriangleright e'), e_2\ (\delta_r \blacktriangleright e')\rangle$

SEL-APP
$\delta_s \blacktriangleright e\ e' \longrightarrow (\delta_s \blacktriangleright e)\ (\delta_s \blacktriangleright e')$

SEL-ABS-EXP
$\delta_s \blacktriangleright \lambda^* v.e \longrightarrow \lambda^* v.(\delta_s \blacktriangleright e)$

SEL-ABS-DIM
d' is fresh $\Rightarrow \delta_s \blacktriangleright \lambda^* d.e \longrightarrow \lambda^* d'.(\delta_s \blacktriangleright [d'/d]e)$

SEL-CHC
$D \neq D' \Rightarrow D_s \blacktriangleright D'\langle e_1, e_2\rangle \longrightarrow D'\langle D_s \blacktriangleright e_1, D_s \blacktriangleright e_2\rangle$

SEL-ANY
d' is fresh $\Rightarrow \delta_s \blacktriangleright (\textbf{any}\ d\ \textbf{from}\ e\ \textbf{in}\ e'\ \textbf{else}\ e'') \longrightarrow \textbf{any}\ d\ \textbf{from}\ \delta_s \blacktriangleright e\ \textbf{in}\ \delta_s \blacktriangleright [d'/d]e'\ \textbf{else}\ \delta_s \blacktriangleright e''$

SEL-THE
$\delta_s \blacktriangleright (\textbf{the}\ \delta'\ \textbf{from}\ e\ \textbf{in}\ e'\ \textbf{else}\ e'') \longrightarrow \textbf{the}\ \delta'\ \textbf{from}\ \delta_s \blacktriangleright e\ \textbf{in}\ \delta_s \blacktriangleright e'\ \textbf{else}\ \delta_s \blacktriangleright e''$

Figure 3 Small-step operational semantics of VPC: reduction and commutation rules.

Reduction rules.

The most basic variation constructs are choice and selection, for introducing and eliminating variation. The elimination of choices by selections is defined by the two Chc-Elim rules, for selecting the left or right alternative of a choice. Observe that selection is only defined on choices with literal dimension names, not dimension variables. This requires that all dimension variables be substituted (see App-Reduce) before choices can be eliminated. Also note that these reductions propagate the selection into the remaining alternative, implementing *choice domination* (see Section 2.2, illustrated by the following reduction sequence.

$$A_r \triangleright A\langle 2, A\langle 3, 4\rangle\rangle \longrightarrow A_r \triangleright A\langle 3, 4\rangle \longrightarrow A_r \triangleright 4 \longrightarrow 4$$

The final step of this sequence depends on the fact that selection on constants (and dimension names) is idempotent, as defined by the two Chc-Idemp rules. Our strategy for eliminating selections will be to continually push selections down to the leaves of an expression, eliminating choices as we go. This will be accomplished mainly by the commutation rules discussed later. However, note that we do not have a selection idempotency rule for selections applied to variables. Such expressions are stuck until a corresponding substitution enables further progress.

The next two rules define function application with aggregating (@-annotated) and non-aggregating functions. The App$^{@}$-Reduce rule is just standard β-reduction. For non-aggregating functions, recall from Section 2.2 that we must instead map the function application over all variants of the argument. In fact, both the LHS and RHS of an application may be variational. If the LHS is a choice, we can apply the App-Chc rule to push the application into both alternatives (and selecting on the RHS to enforce choice domination). We might expect a symmetric rule for pushing an application into a choice on the RHS of an application and then a β-reduction rule for reducing abstractions applied to non-choices. However, this strategy only maps across variation introduced by top-level choices, whereas in general variation may be arbitrarily embedded within an expression, such as in an abstraction body. Therefore, in App-Reduce we define that an unannotated redex is expanded into a **split-any-else** expression that recursively splits e' on every dimension it contains, effectively pushing the application downward into its variants. After we've split on all of the dimensions in e', the **else** branch applies, and the generated expression reduces to standard β-reduction.

Reducing an **any**- or **the**-expression requires querying the variability of the first subexpression, which we call the *scrutinee*. Observe that all of the Any and The reduction rules require the scrutinee to be a value. This forces full reduction of the scrutinee before the expression can be reduced. Ideally, we would reduce the scrutinee to either a choice or a constant, then replace the **any**/**the**-expression by its **then** or **else** branch, respectively. However, this strategy is stymied by the fact that abstractions are also a value form, and abstraction bodies are expressions that may contain choices, but may also contain free variables and so cannot be further reduced to values. This leaves two possible solutions: (1) get stuck when the scrutinee reduces to an abstraction, (2) statically approximate the variability of the reduced scrutinee. Since we want to match against arbitrary values, we choose option (2).

Figure 4 defines an auxiliary function $dims(e)$ that computes a set of dimensions that conservatively approximates the free dimensions of variability in e. A *free* dimension is either a dimension name or a dimension variable that is not bound by an enclosing abstraction in e. Observe in the definition of *dims* that dimension declarations and choices contribute to the

$$
\begin{array}{lll}
dims(\kappa) = dims(v) = \{\} & & dims(e\ e') = dims(e) \cup dims(e') \\
dims(\delta) = \{\delta\} & & dims(\delta\langle e, e'\rangle) = dims(e) \cup dims(e') \cup \{\delta\} \\
dims(\lambda^{*} v.e) = dims(e) & & dims(\delta_s \blacktriangleright e) = dims(e) - \{\delta\} \\
dims(\Lambda^{*} d.e) = dims(e) - \{d\} & &
\end{array}
$$

$$
dims(\textbf{any}\ d\ \textbf{from}\ e\ \textbf{in}\ e'\ \textbf{else}\ e'') = \begin{cases} dims(e') - \{d\} & \text{if } dims(e) \neq \varnothing \\ dims(e'') & \text{otherwise} \end{cases}
$$

$$
dims(\textbf{the}\ \delta\ \textbf{from}\ e\ \textbf{in}\ e'\ \textbf{else}\ e'') = \begin{cases} dims(e') & \text{if } \delta \in dims(e) \\ dims(e'') & \text{otherwise} \end{cases}
$$

Figure 4 Static approximation of dimensions in a VPC expression.

$$
\begin{array}{ll}
(\lambda v.e_1)\ D\langle e_2, e_3\rangle & \\
\longrightarrow \textbf{split}\ D\langle e_2, e_3\rangle\ \textbf{on any}\ d\langle v_\ell, v_r\rangle \to d\langle(\lambda v.e_1)\ e_2, (\lambda v.e_1)\ e_3\rangle\ \textbf{else}\ [D\langle e_2, e_3\rangle/v]e_1 & \\
\qquad\qquad \{\textsc{App-Reduce}\} & \\
\longrightarrow \textbf{any}\ d\ \textbf{from}\ D\langle e_2, e_3\rangle\ \textbf{in split}\ D\langle e_2, e_3\rangle\ \textbf{on}\ d\langle v_\ell, v_r\rangle \to & \\
\qquad d\langle(\lambda v.e_1)\ e_2, (\lambda v.e_1)\ e_3\rangle\ \textbf{else}\ [D\langle e_2, e_3\rangle/v]e_1 \quad \{\textit{expand}\ \textbf{split-any-else}\} & \\
\longrightarrow \textbf{split}\ D\langle e_2, e_3\rangle\ \textbf{on}\ D\langle v_\ell, v_r\rangle \to D\langle(\lambda v.e_1)\ e_2, (\lambda v.e_1)\ e_3\rangle \quad \{\textsc{Any-Reduce-Then}\} & \\
\longrightarrow (\lambda v_\ell\, v_r.\, D\langle(\lambda v.e_1)\ v_\ell, (\lambda v.e_1)\ v_r\rangle)\ (D_\ell \blacktriangleright D\langle e_2, e_3\rangle)\ (D_r \blacktriangleright D\langle e_2, e_3\rangle) & \\
\qquad\qquad \{\textit{expand}\ \textbf{split}\} & \\
\longrightarrow (\lambda v_\ell\, v_r.\, D\langle(\lambda v.e_1)\ v_\ell, (\lambda v.e_1)\ v_r\rangle)\ e_2\ e_3 \quad \{\textsc{Chc-Elim-L}, \textsc{Chc-Elim-R}\} &
\end{array}
$$

Figure 5 Example reduction of $(\lambda v.e_1)\ D\langle e_2, e_3\rangle$.

result set, while selections (which eliminate choices and therefore variability) subtract from it. The dimension abstraction and **any** forms, which declare and scope dimension variables, also subtract from the result set since their dimension variables are not free in e.

Using *dims*, we can define the reduction rules for **the** and **any**. The THE-REDUCE rules simply check to see whether the specified dimension is present in $dims(\underline{e})$ and reduce to either the **then** or **else** branch accordingly. An **any**-expression reduces to the **else** branch if $dims(\underline{e})$ is empty (ANY-REDUCE-ELSE), but if it's not empty we must pick a dimension to bind to d (in ANY-REDUCE-THEN). The dimension we pick is based on an ordering relation ($\prec$) on dimensions: for any dimension name D and dimension variable d', $D \prec d'$; when comparing two dimension names or two dimension variables, $\prec$ is the lexicographic ordering of their names. The function $\min(\overline{\delta})$ returns the minimum $\delta \in \overline{\delta}$ according to this ordering relation. (We use the overbar to denote lists.)

The requirement that the scrutinee of an **any**/**the**-expression be fully evaluated is needed to ensure confluence since *dims* is an approximation of the actual variation in e. To illustrate, consider the expression[2] **ifvar** $(\Lambda^{\circledast} x.2)\ A\langle 3, 4\rangle$ **then** 5 **else** 6. Forcing evaluation of the scrutinee first yields reduces the expression to **ifvar** 2 **then** 5 **else** 6 and then to 6 since $dims(2) = \varnothing$. However, if we allowed also reducing **any**/**the** immediately, the expression reduces to 5 since $dims((\Lambda^{\circledast} x.2)\ A\langle 3, 4\rangle) = \{A\}$.

Finally, the LET-REDUCE rule implements a recursive **let** by a nested substitution.

[2] Recall that **ifvar** expands to an **any**–**then**–**else** expression.

Selection commutation rules.

The App-Chc rule described in the previous section pushes applications into choices on the LHS in order to setup future reductions via the App-Reduce or App$^@$-Reduce. Similarly, the eight Sel-* rules push selections downward to setup the elimination of matching choices via the Chc-Elim rules, and eventually to reduction at the leaves via the Sel-Idemp rules.

The Sel-App and Sel-Let rules straightforwardly push selections into applications and **let**-expressions, respectively. The two Sel-Abs rules push selections into abstractions. In these rules we distinguish between expression abstractions (where the variable is in the namespace v) and dimension abstractions (in namespace d). However, both rules apply to both annotated and unannotated abstractions. We make this explicit by writing $\lambda^* v.e$ and assume that the annotations are preserved by the transformation. Note that the Sel-Abs-Dim rule renames the bound dimension variable to preemptively avoid the capture of the selected dimension δ, which may also be a dimension variable.

The Sel-Chc rule pushes selections into choices. This rule can only be applied if both dimensions (the one being selected, and the one referenced by the choice) have been resolved to dimension names. Selection on a choice involving a dimension variable is stuck until dimension substitution enables further progress. To illustrate why this is necessary, consider the expression $d_\ell \blacktriangleright d'\langle e_1, e_2\rangle$. At first it may seem that this expression is equivalent to $d'\langle d_\ell \blacktriangleright e_1, d_\ell \blacktriangleright e_2\rangle$ since the dimension variables have different names. However, it might be that d and d' are substituted by the same dimension name, in which case we should have eliminated the choice rather than pushing the selection into it.

Finally, the Sel-Any and Sel-The rules push selections into **any**- and **the**-expressions. Since **any** binds dimension names, the Sel-Any rule renames to avoid dimension variable capture.

Congruence rules.

For space reasons, we relegate to the long version of this paper [18] the congruence rules, which complete the reduction relation. They are entirely straightforward. We give them names like In-Any-Then for reducing the **then**-branch of an **any**-expression, and In-Chc-L for reducing the left alternative of a choice. The most interesting feature of the congruence rules are two rules that are *omitted*—we do not define In-Abs or In-Sel rules for reducing the body of abstractions or selections, respectively. The reason these are omitted is to avoid prematurely reducing **any**- and **the**-expressions. When an **any**/**the** occurs in the body of an abstraction, future variable substitutions may affect the variability of the scrutinee, and so we should not reduce it until all free variables have been substituted. Conversely, when an **any**/**the** occurs in the body of a selection, the scrutinee may contain *more* variation than it will after the selections have been pushed all the way down. Thus, reducing within abstractions and selections would lead to semantics that is not confluent.

Confluence.

The main result for the operational semantics is that the reduction relation is confluent, captured in the following lemma. (For a proof sketch, see [18]).

▸ **Lemma 1** (Local confluence). $e \longrightarrow e' \wedge e \longrightarrow e'' \implies \exists e'''.\ e' \longrightarrow^* e''' \wedge e'' \longrightarrow^* e'''$

$$
\begin{array}{lcll}
\tau & ::= & \gamma \mid \alpha \mid \tau \to \tau & (\textit{plain types}) \\
\phi & ::= & \tau \mid \delta\langle\phi,\phi\rangle \mid \phi \to \phi \mid \delta & (\textit{variational types}) \\
\sigma & ::= & \forall\overline{\alpha}\overline{d}.\phi \mid \forall\overline{\alpha}\overline{d}.C \Rightarrow \phi & (\textit{variational schemas}) \\
C & ::= & \phi \equiv \phi \mid C \wedge C \mid \delta\langle C,C\rangle \mid \exists\overline{\alpha}\overline{d}.C \mid \delta \in \phi \Rightarrow \delta{\downarrow}\phi & (\textit{constraints})
\end{array}
$$

Figure 6 Type Syntax.

4 Type System

This section presents a type system for VPC. In Section 4.1 we define the syntax of the type language, and in Section 4.2 we define the typing relationship through a set of typing rules.

4.1 Syntax

Types are stratified into three layers (see Figure 6). First, we use τ to range over *plain types*, which don't contain variations. Plain types include type constants γ, type variables α, and function types. Second, we use ϕ to range over variational types. A choice type $\delta\langle\phi_1,\phi_2\rangle$ allows us to represent type variation. By including dimensions δ, which are expressions, in type syntax we obtain a lightweight form of dependent types. Finally, we have type schemas, which are variational types universally quantified over type variables α and dimension variables d. We refer to type schemas of the form $\forall d.\phi$ as *dimension-polymorphic types*. Note that we don't explicitly represent choice types containing polymorphic types since they can be transformed to type schemas, as shown in [10].[3] With this type syntax we can characterize types for expressions manipulating dimensions and choices. Consider, for example, the function *poly* that takes a dimension and creates a choice in that dimension.

$$
\begin{aligned}
&\mathit{poly} : \forall d.d \to d\langle \texttt{Int}, \texttt{Bool}\rangle \\
&\mathit{poly} = \lambda d.d\langle 2, \texttt{True}\rangle
\end{aligned}
$$

In this example, the type precisely describes the intention of *poly*. While we can instantiate *poly* with any dimension, we may want to restrict dimensions that can be used to instantiate dimension-polymorphic types in general. We use the following expression *bounded* to illustrate this point.

$$\mathit{bounded} = \lambda^{\circledR} e.\textbf{any}\ d\ \textbf{from}\ e\ \textbf{in}\ d\langle 2, \texttt{True}\rangle\ \textbf{else}\ \texttt{undefined}$$

Here *bounded* finds a dimension in the expression supplied as argument based on some ordering that is discussed in Section 3. What should be the type of *bounded*? The type $\forall d.\phi \to d\langle \texttt{Int}, \texttt{Bool}\rangle$ is not precise enough since it can be instantiated with any dimension, even one absent in the input expression. To address this issue, we introduce constrained types that allow us to attach constraints to polymorphic types. Constraints specify the valid ways of instantiating polymorphic variables. To represent the type for *bounded*, we introduce the constraint $d{\downarrow}\phi$, which requires that d must be the particular dimension in ϕ according to the defined ordering. For example, for $\phi = B\langle A\langle \texttt{Int}, \texttt{Bool}\rangle, \texttt{Bool}\rangle$, we have $A{\downarrow}\phi$. With this constraint, we can write the type for *bounded* as follows.

$$\mathit{bounded} : \forall d.d{\downarrow}\phi \Rightarrow \phi \to d\langle \texttt{Int}, \texttt{Bool}\rangle$$

[3] E.g., we don't deal with $A\langle \forall\alpha.\alpha \to \alpha, \forall\alpha.\alpha \to \texttt{Int}\rangle$ since we can transform it to $\forall\alpha,\beta.A\langle \alpha \to \alpha, \beta \to \texttt{Int}\rangle$.

$$\text{E1}\quad C \Vdash C \qquad \text{E2}\quad C \wedge C_1 \Vdash C_1 \qquad \text{E3}\quad C \Vdash \exists \overline{\alpha}\overline{d}.C$$

$$\text{E4}\ \frac{C_1 \Vdash C_2 \qquad C_2 \Vdash C_3}{C_1 \Vdash C_3} \qquad \text{E5}\ \frac{C_1 \Vdash C_2}{\theta(C_1) \Vdash \theta(C_2)}$$

$$\text{E6}\ \frac{C_1 \Vdash C_2}{\exists \overline{\alpha}\overline{d}.C_1 \Vdash \exists \overline{\alpha}\overline{d}.C_2} \qquad \text{E7}\ \frac{C_1 \Vdash C_2 \qquad C_3 \Vdash C_4}{\delta\langle C_1, C_3\rangle \Vdash \delta\langle C_2, C_4\rangle}$$

$$\text{E8}\ \frac{FD(\phi_1) = \varnothing \qquad FV(\phi_1) = \varnothing \qquad \delta \notin dims(\phi_1)}{\Vdash \delta \in \phi_1 \Rightarrow \delta{\downarrow}\phi} \qquad \text{E9}\ \frac{\delta \in dims(\phi_1) \qquad \delta = \min(dims(\phi))}{\Vdash \delta \in \phi_1 \Rightarrow \delta{\downarrow}\phi}$$

$$\text{T1}\quad \phi_1 \equiv \phi_2 \Vdash \phi_2 \equiv \phi_1 \qquad \text{T2}\quad \phi_1 \equiv \phi_2 \wedge \phi_2 \equiv \phi_3 \Vdash \phi_1 \equiv \phi_3 \qquad \text{T3}\quad \Vdash \phi \equiv \phi$$

$$\text{T4}\quad \phi_1 \equiv \phi_2 \Vdash \phi[\phi_1] \equiv \phi[\phi_2] \qquad \text{T5}\quad \Vdash \delta\langle\phi, \phi\rangle \equiv \phi \qquad \text{T6}\quad \Vdash \delta\langle\phi_1, \phi_2\rangle \equiv \delta\langle\lfloor\phi_1\rfloor_{\delta_\ell}, \lfloor\phi_2\rfloor_{\delta_r}\rangle$$

Figure 7 Entailment relation of constraints.

This type states that for any given argument, there is at most one valid instantiation of d. Note that the constraint $d{\downarrow}\phi$ must be satisfied only if d appears in the result type, as it does in *bounded*. In general, the conditional constraint $\delta \in \phi_2 \Rightarrow \delta{\downarrow}\phi_1$ means that $\delta{\downarrow}\phi_1$ must be satisfied only if $\delta \in \phi_2$ is satisfied.

We define other constraints in Figure 6 and use the metavariable C to range over constraints. The constraint $\phi_1 \equiv \phi_2$ requires that types ϕ_1 and ϕ_2 be equivalent, which is more flexible than type equality. The "variational constraint" $\delta\langle C_1, C_2\rangle$ expresses a choice between constraint C_1 or C_2, depending on dimension δ. The constraint forms $C_1 \wedge C_2$ and $\exists \overline{\alpha}\overline{d}.C$ have the conventional meanings.

We now turn to the relations among constraints. We use the entailment relation $C_1 \Vdash C_2$ to denote that if C_1 is satisfied then C_2 must also be satisfied. The rules E1, E2, and E4 are standard in constrained type systems. The rules E3 and E6 specify that existential quantifiers hide information of constraints and thus quantified constraints are easier to satisfy. The constraint E5 states that the entailment relation is stable under substitution, where θ is a mapping of type variables to types and dimension variables to dimensions. Rule E7 deals with variational constraints and is obvious.

Finally, rules E8 and E9 describe two ways to satisfy the constraint $\delta \in \phi_1 \Rightarrow \delta{\downarrow}\phi$. The first applies when δ doesn't occur in ϕ_1, which we can conclude only when ϕ_1 doesn't contain dimension variables or type variables because they may be substituted with other variational types. The operation $FD(\phi)$ is defined as follows.

$$\begin{aligned} FD(d\langle\phi_1, \phi_2\rangle) &= \{d\} \cup FD(\phi_1) \cup FD(\phi_2) \\ FD(D\langle\phi_1, \phi_2\rangle) &= FD(\phi_1) \cup FD(\phi_2) \\ FD(\phi_1 \to \phi_2) &= FD(\phi_1) \cup FD(\phi_2) \\ FD(d) &= \{d\} \qquad FD(D) = FD(\tau) = \varnothing \end{aligned}$$

The operation $dims(\phi)$ is defined similarly but collecting all δs. The second situation is handled by E9, which requires that $\delta \in dims(\phi_1)$ and that δ is $\min(dims(\phi_2))$.

The second part of Figure 7 presents the entailment relation between type equivalence relations. The rules T1 through T3 state that type equivalence is reflexive, symmetric, and

$$\boxed{C;\Gamma;\Delta \vdash e : \phi}$$

$$\textsc{Con}\ \frac{\kappa \text{ of type } \gamma}{C;\Gamma;\Delta \vdash \kappa : \gamma} \qquad \textsc{Var}\ \frac{\Gamma(v) = \forall \overline{\alpha}\overline{d}.C \Rightarrow \phi \qquad C' \Vdash C}{C';\Gamma;\Delta \vdash v : \phi} \qquad \textsc{Abs}\ \frac{C;\Gamma,(v,\phi_1);\Delta \vdash e : \phi}{C;\Gamma;\Delta \vdash \lambda v.e : \phi_1 \to \phi}$$

$$\textsc{App}\ \frac{C;\Gamma;\Delta \vdash e_1 : \phi_1 \qquad C;\Gamma;\Delta \vdash e_2 : \phi_2 \qquad C \Vdash \phi_1 \equiv \phi_2 \to \phi}{C;\Gamma;\Delta \vdash e_1\ e_2 : \phi} \qquad \textsc{AbsD}\ \frac{C;\Gamma;\Delta,(d,\delta) \vdash e : \phi}{C;\Gamma;\Delta \vdash \lambda d.e : \delta \to \phi}$$

$$\textsc{Let}\ \frac{\begin{array}{c} C_1 \wedge C_2;\Gamma,(v,\phi);\Delta \vdash e_1 : \phi \\ \overline{d} \# FD(C_2) \cup FD(\Gamma) \qquad \overline{\alpha} \# FV(C_2) \cup FV(\Gamma) \qquad C_2;\Gamma,(v,\forall \overline{\alpha}\overline{d}.C_1 \Rightarrow \phi);\Delta \vdash e_2 : \phi_2 \end{array}}{C_2 \wedge \exists \overline{\alpha}\overline{d}.C_1;\Gamma;\Delta \vdash \textbf{let } v = e_1 \textbf{ in } e_2 : \phi_2}$$

$$\textsc{Dim}\ \frac{\Delta(\delta) = \delta_1}{C;\Gamma;\Delta \vdash \delta : \delta_1} \qquad \textsc{Chc}\ \frac{C_1;\Gamma;\Delta \vdash e_1 : \phi_1 \qquad C_2;\Gamma;\Delta \vdash e_2 : \phi_2 \qquad \Delta(\delta_1) = \delta}{\delta\langle C_1, C_2\rangle;\Gamma;\Delta \vdash \delta_1\langle e_1, e_2\rangle : \delta\langle \phi_1, \phi_2\rangle}$$

$$\textsc{Sel}\ \frac{C;\Gamma;\Delta \vdash e : \phi \qquad \Delta(\delta) = \delta_1 \qquad C \Vdash \delta_1\langle \phi_\ell, \phi_r\rangle \equiv \phi \qquad \begin{cases} C \Vdash \alpha \equiv \phi_\ell & s = \ell \\ C \Vdash \alpha \equiv \phi_r & s = r \end{cases}}{C;\Gamma;\Delta \vdash \delta_s \blacktriangleright e : \alpha}$$

$$\textsc{Any}\ \frac{\begin{array}{c} C;\Gamma;\Delta \vdash e_1 : \phi_1 \\ C;\Gamma;\Delta,(d,d) \vdash e_2 : \phi_2 \qquad C;\Gamma;\Delta \vdash e_3 : \phi_3 \qquad C \Vdash \phi_2 \equiv \phi_3 \qquad C \Vdash d \in \phi_3 \Rightarrow d{\downarrow}\phi_1 \end{array}}{C;\Gamma;\Delta \vdash \textbf{any } d \textbf{ from } e_1 \textbf{ in } e_2 \textbf{ else } e_3 : \phi_3}$$

$$\textsc{The}\ \frac{C;\Gamma;\Delta \vdash e_1 : \phi_1 \qquad \Delta(\delta) = \delta_1 \qquad C;\Gamma;\Delta \vdash e_2 : \phi_2 \qquad C;\Gamma;\Delta \vdash e_3 : \phi_3 \qquad C \Vdash \phi_2 \equiv \phi_3}{C;\Gamma;\Delta \vdash \textbf{the } \delta \textbf{ from } e_1 \textbf{ in } e_2 \textbf{ else } e_3 : \phi_3}$$

Figure 8 Typing rules.

transitive. Rule T4 describes that equivalence is congruent with respect to an arbitrary shared context. The rules T5 and T6 state that idempotent choices and dead alternatives may be eliminated. The operation $\lfloor \phi_1 \rfloor_{\delta_\ell}$ replaces each occurrence of a choice in dimension δ within ϕ_1 with its left alternative [17].

4.2 Typing Rules

We use the judgment $C;\Gamma;\Delta \vdash e : \phi$ to denote that under the constraint C, the type environment Γ, and the dimension environment Δ, the expression e has the type ϕ. The environment Δ that collects all the assumptions for bound dimension variables that are visible for e is a mapping from dimension variables d to dimensions δ. We frequently use the operation $\Delta(\delta)$,

which is defined as follows.

$$\Delta(\delta) = \begin{cases} D & \text{if } \delta = D \\ \delta_1 & \text{if } \delta = d \wedge (d, \delta_1) \in \Delta \\ \text{undefined} & \text{if } \delta = d \wedge d \notin dom(\Delta) \end{cases}$$

The first three rules (Con, Var, and Abs) are standard. The rule App for function application relaxes the usual equality constraint between the function parameter and argument and requires only that they be equivalent (not equal). The rule AbsD is similar to Abs except that it abstracts over dimension variables. To type dimension abstractions, we assume a binding for (d, δ) in Δ, then type the body.

The Let rule deals with recursive **let** expressions. Note that we allow polymorphism over both dimension variables and type variables. The overall typing process is standard. However, note that we only attach the constraint (C_1) that specifies requirements for $\overline{\alpha}$ and $\overline{d}$ to the type ϕ assumed for v. Since the variable v will probably be referred to many times in the body, this helps to control the size of the constraint aggregated during the typing process. As been noted earlier [40, 44, 13], the size of constraints has a huge impact on the performance of type checkers. We therefore regard this optimization as necessary. Another subtlety is that we keep the constraint $\exists\overline{\alpha}\overline{d}.C_1$ in the result typing constraint. This constraint increases the precision of the type system. For a discussion, see [40, 44].

Rule Dim consults the binding for δ in Δ. From the definition of $\Delta(\delta)$ it follows that $C; \Gamma; \Delta \vdash D : D$ and $C; \Gamma; \Delta \vdash d : D$ if $(d, D) \in \Delta$. The rule Chc for choice construction expresses the idea that the typing process can be composed over choice creation. Specifically, if e_1 and e_2 are typed under the constraints C_1 and C_2, respectively, then $\delta_1\langle e_1, e_2\rangle$ is well typed under the variational constraint $\delta\langle C_1, C_2\rangle$, where $\delta = \Delta(\delta_1)$. To type a selection, we first retrieve the type ϕ_1 of the expression being selected. Then, based on the selector given, the constraint must ensure that the result type is equivalent to the corresponding alternative of ϕ_1. This idea is formalized in rule Sel.

To type an **any** expression, we first derive the type ϕ_1 for e_1. Then we have to consider two different cases. First, if ϕ_1 contains a dimension, we type the **then** branch e_2 by extending Δ with (d, d). Otherwise, if ϕ_1 doesn't contain any dimension, then the else branch e_3 is typed. The language requires that both branches have the same type. Another constraint that needs to be satisfied is that if the result type of the expression refers to d, then d must appear in ϕ_1 and must be the smallest dimension according to the ordering of dimensions defined in Section 3. This is expressed through the entailment relation $C \Vdash d \in \phi_3 \Rightarrow d{\downarrow}\phi_1$. The rule Any specifies this typing process.

Note that the result will be incorrect if we drop the condition and use the constraint entailment $C \Vdash d{\downarrow}\phi_1$. To illustrate, consider the expression `vmin` introduced in Section 2.4. Although we will not describe the formal reasoning process, we briefly discuss how the type for `vmin` can be derived. First, since `vmin` is a recursive function, we assume its type is $\alpha \to \beta$. Next, the call of `min` of result type β forces β to be `Int` and the return type of the **then** branch to be `Int`. As both branches of **any** need to have the same type, the **else** branch of type α also has the type `Int`. Thus, both α and β are `Int`. If the **any** expression introduces the constraint $d{\downarrow}$`Int`, we observe that the constraint can never be satisfied since `Int` doesn't contain any dimension. However, if we introduce the constraint $d \in$ `Int` $\Rightarrow d{\downarrow}$`Int`, then the constraint can immediately be removed according to rule E8 in Figure 7.

The idea of typing **the** expressions is very similar to that of typing **any** expressions. Both branches of **the** also need to be equivalent. The main difference is that this rule doesn't place a constraint on the dimension δ, as can be seen from the rule The. The reason is that δ must be bound by a dimension abstraction. Note that while other judgments in The doesn't

$$\text{I-App}\ \frac{C_1;\Gamma;\Delta \vdash_I e_1 : \alpha_1 \qquad C_2;\Gamma;\Delta \vdash_I e_2 : \alpha_2 \qquad \alpha_3 \text{ fresh}}{\exists\alpha_1\alpha_2.(C_1 \wedge C_2 \wedge \alpha_1 \equiv \alpha_2 \to \alpha_3);\Gamma;\Delta \vdash_I e_1\ e_2 : \alpha_3}$$

$$\text{I-Any}\ \frac{\begin{array}{c} C_1;\Gamma;\Delta \vdash_I e_1 : \alpha_1 \\ C_2;\Gamma;\Delta,(d,d) \vdash_I e_2 : \alpha_2 \qquad C_3;\Gamma;\Delta \vdash e_3 : \alpha_3 \qquad C_4 = (d \in \alpha_3 \Rightarrow d{\downarrow}\alpha_1) \end{array}}{\exists\alpha_1\alpha_2 d.(C_1 \wedge C_2 \wedge C_3 \wedge \alpha_2 \equiv \alpha_3 \wedge C_4);\Gamma;\Delta \vdash_I \textbf{any}\ d\ \textbf{from}\ e_1\ \textbf{in}\ e_2\ \textbf{else}\ e_3 : \alpha_3}$$

Figure 9 Constraint generation for VPC.

use δ_1, we write $\Delta(\delta) = \delta_1$ to ensure that δ is a dimension constant or is bound in Δ.

Our type system is sound with respect to the operational semantics defined in Section 3.2. Specifically, the following two properties hold.

▸ **Theorem 2** (Progress). *If $C;\Gamma;\varnothing \vdash e : \phi$ and Γ contains information for constants only and $FV(\Gamma) = \varnothing$, and $\Vdash C$, then e is a value or there is some e' such that $e \longrightarrow e'$.*

▸ **Theorem 3** (Preservation). *If $C;\Gamma;\Delta \vdash e : \phi$ and $e \longrightarrow e'$, then $C;\Gamma;\Delta \vdash e' : \phi$.*

Progress can be proved by induction on the typing derivation, and preservation by induction on the reduction relation.

5 Constraint Generation

The type inference process consists of two steps: constraint generation and constraint solving. This section presents constraint generation and investigates its properties. Constraint solving will be presented in Section 6.

The goal of constraint generation is to collect constraints for a specific expression e under the type environment Γ and dimension environment Δ. Figure 9 presents constraint generation rules for function applications and **any** expressions. The full set of constraint generation rules is presented in [18]. The rules define the judgment $C;\Gamma;\Delta \vdash_I e : \alpha$, which means that given e, Γ, and Δ, the constraint C will be collected and the result type is α.

The constraint generation rules are derived from the typing rules in Figure 8. The constraints for a given expression e are a combination of the constraints of its subexpressions and constraints relating the types of its subexpressions. For example, in the I-App rule, the constraints for $e_1\ e_2$ are obtained as the conjunction of the constraints C_1, C_2, and $\alpha_1 \equiv \alpha_2 \to \alpha_3$, where C_1 and C_2 are the constraints for e_1 and e_2, respectively, and the constraint $\alpha_1 \equiv \alpha_2 \to \alpha_3$ relates the types of e_1, e_2, and $e_1\ e_2$.

An important observation is that the constraint generation rules collect exactly the constraints specified in the typing relation in Figure 8—they do not forget or introduce new constraints. This leads to the following soundness and completeness theorems for constraint generation.

▸ **Theorem 4** (Soundness of constraint generation). *If $C;\Gamma;\Delta \vdash_I e : \alpha$, then $C;\Gamma;\Delta \vdash e : \alpha$.*

▸ **Theorem 5** (Completeness and principality of constraint generation). *If $C;\Gamma;\Delta \vdash e : \phi$, then $C';\Gamma;\Delta \vdash_I e : \alpha$ with $C \Vdash \exists\alpha.C'$ and $C \wedge C' \Vdash \theta(\alpha) \equiv \phi$ for some substitution θ.*

Theorem 4 can be proved by induction over the constraint generation rules, Theorem 5 by induction over the typing rules.

Based on the constraint generation rules, the following set of constraints (reformatted for readability) will be generated for the expression *bounded* $A\langle 2, 3\rangle$ (where *bounded* is the function introduced in Section 4.1). We refer to these constraints collectively as $\mathcal{C}_1$ (and use $\mathcal{C}$ to range over constraint sets henceforth). Here α and α_1 represent the return type and argument type of *bounded*, respectively.

$$C_1 : \alpha \equiv d\langle \texttt{Int}, \texttt{Bool}\rangle \qquad C_2 : d \in d\langle \texttt{Int}, \texttt{Bool}\rangle \Rightarrow d{\downarrow}\alpha_1 \qquad C_3 : \alpha_1 \equiv \alpha_2 \qquad C_4 : \alpha_2 \equiv A\langle \texttt{Int}, \texttt{Int}\rangle$$

The constraint set $\mathcal{C}_2$ for the application *bounded* `2` has the same first three constraints as $\mathcal{C}_1$ and has a different last constraint C_5.

$$C_1 : \alpha \equiv d\langle \texttt{Int}, \texttt{Bool}\rangle \qquad C_2 : d \in d\langle \texttt{Int}, \texttt{Bool}\rangle \Rightarrow d{\downarrow}\alpha_1 \qquad C_3 : \alpha_1 \equiv \alpha_2 \qquad C_5 : \alpha_2 \equiv \texttt{Int}$$

6 Constraint Solving

While generating constraints from the typing is straightforward, solving such constraints is more challenging. Section 6.1 discusses these challenges, and Section 6.2 presents a constraint solver.

6.1 Constraint Solving Challenges

There are three main challenges to address. First, rule T5 in Figure 7 specifies that we can always simplify a type by eliminating idempotent choices. Moreover, doing so can significantly improve the performance of type inference algorithms in practice, as has been shown in [17]. However, eagerly eliminating all idempotent choices can render some satisfiable constraints unsatisfiable. Consider, for example, the constraint set $\mathcal{C}_1$ generated in Section 5. If we solve C_3 and C_4 with the substitution $\theta = \{\alpha_2 \mapsto \texttt{Int}, \alpha_1 \mapsto \texttt{Int}\}$, then C_2 cannot be satisfied since $d{\downarrow}\texttt{Int}$ fails because `Int` doesn't contain any dimension. This is undesirable since the expression that generates $\mathcal{C}_1$ is well typed. However, if we instead use the substitution $\theta' = \{\alpha_2 \mapsto A\langle \texttt{Int}, \texttt{Int}\rangle, \alpha_1 \mapsto A\langle \texttt{Int}, \texttt{Int}\rangle\}$, then C_2 can be solved by mapping d to A. Finally, solving C_1 gives us $A\langle \texttt{Int}, \texttt{Bool}\rangle$ as the result type of *bounded* $A\langle 2, 3\rangle$. This result matches our expectation since the expression is indeed well typed.

On the other hand, we shouldn't introduce gratuitous idempotent choices, because those might allow solving some unsatisfiable constraints. Consider, for example, the constraint set $\mathcal{C}_2$ from Section 5. We again have both θ and θ' as potential solutions for $\mathcal{C}_2$. Since the expression that generates $\mathcal{C}_2$ is ill typed, only θ leads to the expected result since it makes C_2 unsolvable.

The second challenge regards the kind of information produced through constraint solving. This is straightforward for constraints on type variables. For example, in the case of $A\langle \texttt{Int}, \alpha\rangle \equiv A\langle \alpha, \texttt{Bool}\rangle$, we simply find a substitution for α. But what do we do about $d_1\langle \texttt{Int}, \texttt{Bool}\rangle \equiv d_2\langle \texttt{Int}, \texttt{Bool}\rangle$? At best, this constraint allows us to derive that $d_1 = d_2$. However, in general such constraints are undecidable. For example, consider $A\langle \texttt{Int}, \alpha\rangle \equiv d\langle \texttt{Int}, \beta\rangle$. Both $\{d \mapsto A, \alpha \mapsto \texttt{Int}, \beta \mapsto \texttt{Int}\}$ and $\{d \mapsto B, \alpha \mapsto \texttt{Int}, \beta \mapsto \texttt{Int}\}$ are possible substitutions but neither is more general than the other.

The third challenge is that, unlike in normal constraint solving [40], a set of constraints cannot be solved in one iteration. Consider, for example, the following constraint set $\mathcal{C}_3$.

$$C_6 : d_1\langle \texttt{Int}, \texttt{Bool}\rangle \equiv d_2\langle \texttt{Int}, \texttt{Bool}\rangle \qquad\qquad C_7 : d_1 \equiv A \wedge d_2 \equiv B$$

For the same reason as in the previous paragraph*, solving C_6 directly won't provide useful information. We can solve C_7, however, with $d_1 \mapsto A$ and $d_2 \mapsto B$. By substituting d_1 with A

$$
\begin{aligned}
\mathit{noelim}(\phi_1 \equiv \phi_2, \mathcal{K}) &= \begin{cases} \mathcal{K} & \mathcal{K} \cap FV(\phi_1, \phi_2) = \varnothing \\ \mathcal{K} \cup FV(\phi_1, \phi_2) & \text{otherwise} \end{cases} \\
\mathit{noelim}(\exists \overline{\alpha}\overline{d}.C, \mathcal{K}) &= \mathit{noelim}(C, \mathcal{K} - \overline{\alpha}) \\
\mathit{noelim}(C_1 \wedge C_2, \mathcal{K}) &= \mathit{noelim}(\delta\langle C_1, C_2\rangle, \mathcal{K}) = \mathit{noelim}(C_1, \mathcal{K}) \cup \mathit{noelim}(C_2, \mathcal{K}) \\
\mathit{noelim}(\delta \in \phi_1 \Rightarrow \delta{\downarrow}\phi, \mathcal{K}) &= \mathit{noelim}(\delta{\downarrow}\phi, \mathcal{K}) = \mathcal{K} \cup FV(\phi) \\
\mathit{noelim}(\mathcal{C}, \mathcal{K}) &= \begin{cases} \mathcal{K} & \mathcal{K} = \mathcal{K}' \\ \mathit{noelim}(\mathcal{C}, \mathcal{K}') & \text{otherwise} \end{cases} \qquad \text{where } \mathcal{K}' = \bigcup_{C \in \mathcal{C}} \mathit{noelim}(C, \mathcal{K})
\end{aligned}
$$

Figure 10 Keeping idempotent choices.

and d_2 with B in C_6, we can now find that C_6 is unsolvable. Overall, we need two iterations to determine that $\mathcal{C}_3$ is unsolvable.

To address the first challenge, we require that idempotent choices are not reduced when solving certain constraints. To address the second challenge and make constraint solving decidable, we defer computing solutions for dimension variables when they are used as choice constructors. Finally, to address the third challenge, we use iterative constraint solving, that is we solve constraints in multiple iterations and stop when no further progress can be made.

To simplify the presentation, in addition to the constraint forms in Figure 6, we introduce a new constraint form, $d{\downarrow}\phi$, which is satisfied if $d = \min(\mathit{dims}(\phi))$. We also use `true` to denote a constraint that is trivially satisfied. We define the auxiliary function $\mathit{noelim}(\mathcal{C}, \mathcal{K})$ in Figure 10 to decide when idempotent choices shouldn't be removed. Given $\mathcal{C}$, the iterative function $\mathit{noelim}(\mathcal{C}, \{\})$ returns the set of type variables for which we must preserve idempotent choices. Intuitively, $\mathit{noelim}(\mathcal{C}, \{\})$ includes (1) the type variables found in variational types ϕ for constraints of the form $\delta \in \phi_1 \Rightarrow \delta{\downarrow}\phi$ or $\delta{\downarrow}\phi$ and (2) the type variables that share a constraint with some other type variables in $\mathit{noelim}(\mathcal{C}, \{\})$. $\mathit{noelim}(\mathcal{C}, \{\})$ is complete since it handles all different forms of the constraint.

Given this operation, we can compute $\mathit{noelim}(\mathcal{C}_1, \{\}) = \{\alpha_1, \alpha_2\}$, where $\mathcal{C}_1$ is from Section 5. Note that α is not included in the resulting set since it shares no constraint with either α_1 or α_2.

6.2 A Constraint Solver

We can now implement a constraint solver $\mathcal{S}$. Given a constraint set $\mathcal{C}$ and a mapping θ, $\mathcal{S}$ solves iteratively until no progress can be made, which is detected when $\mathcal{C} = \mathcal{C}'$ where $\mathcal{C}'$ is the residual constraint set of the current iteration. Note that a new $\mathit{noelim}(\mathcal{C}, \{\})$ is computed and passed to $\mathcal{U}'$ at the beginning of each iteration since constraints may have been changed during the previous iteration.

$$
\mathcal{S}(\mathcal{C}, \theta) = \begin{cases} (\mathcal{C}', \theta') & \mathcal{C} = \mathcal{C}' \\ \mathcal{S}(\mathcal{C}', \theta') & \text{otherwise} \end{cases} \qquad \text{where } (\mathcal{C}', \theta') = \mathcal{U}'(\mathcal{C}, \theta, \mathit{noelim}(\mathcal{C}, \{\}))
$$

The real work of solving constraints is performed by $\mathcal{U}'$, which in turn calls $\mathcal{U}$ to solve each constraint individually. Both $\mathcal{U}'$ and the main part of $\mathcal{U}$ are given in Figure 11. Several simple cases, such as unifying two plain types and unifying two dimension variables, for $\mathcal{U}$ are omitted. Given $\mathcal{C}$, $\mathcal{U}'$ will return the same result if the constraints are solved in different orderings. However, if $\mathcal{C}$ fails to solve, $\mathcal{U}'$ will fail on different constraints considering

$$\mathcal{U}'(\{C_1,\ldots,C_n\},\theta,\mathcal{K}) = \textbf{let } (C_1',\theta_1') = \mathcal{U}(C_1,\theta,\mathcal{K}) \\ (\mathcal{C}',\theta') = \mathcal{U}'(\theta_1'(\{C_2,\ldots,C_n\}),\theta_1',\mathcal{K}) \\ \textbf{in } (\{C_1'\}\cup\mathcal{C}',\theta')$$

(a) $\mathcal{U}(\exists\alpha d.C,\theta,\mathcal{K}) = \textbf{let } (C_1,\theta_1) = \mathcal{U}(C,\theta,\mathcal{K}) \textbf{ in } (\exists\alpha d.C_1,\theta_1\backslash\{\alpha,d\})$

(b) $\mathcal{U}(\delta \in \phi_1 \Rightarrow \delta{\downarrow}\phi_2,\theta,\mathcal{K})$
- $\mid \delta \in \mathit{dims}(\theta(\phi_1)) = \mathcal{U}(\delta{\downarrow}\phi_2,\theta,\mathcal{K})$
- $\mid \mathit{FD}(\theta(\phi_1)) == \varnothing \wedge \mathit{FV}(\theta(\phi_1)) == \varnothing \wedge \delta \notin \mathit{dims}(\theta(\phi_1)) = (\mathit{true},\theta)$
- $\mid$ **otherwise** $= (\delta \in \phi_1 \Rightarrow \delta{\downarrow}\phi_2,\theta)$

(c) $\mathcal{U}(\delta{\downarrow}\phi_2,\theta,\mathcal{K})$
- $\mid \mathit{FD}(\theta(\phi_2)) \neq \varnothing \vee \mathit{FV}(\theta(\phi_2)) \neq \varnothing = (\delta{\downarrow}\phi_2,\theta)$
- $\mid \mathit{dims}(\theta(\phi_2)) == \varnothing =$ fail
- $\mid \delta == d = (\mathit{true},\{d \mapsto \min(\mathit{dims}(\theta(\phi_2)))\}\circ\theta)$
- $\mid \delta \neq \min(\mathit{dims}(\theta(\phi_2)))$ = fail
- $\mid$ **otherwise** $= (\mathit{true},\theta)$

(d) $\mathcal{U}(D_1\langle\phi_1,\phi_2\rangle \equiv D_2\langle\phi_3,\phi_4\rangle,\theta,\mathcal{K}) =$
let $(\theta_1,r) = \mathit{vunify}(D_1\langle\phi_1,\phi_2\rangle, D_2\langle\phi_3,\phi_4\rangle)$ **in if** r **then** $(\mathit{true},\theta_1\circ\theta)$ **else** fail

(e) $\mathcal{U}(\alpha \equiv \phi,\theta,\mathcal{K}) =$ **if** $\alpha \in \mathcal{K}$ **then** $(\mathit{true},\{\alpha \mapsto \phi\}\circ\theta)$ **else** $(\mathit{true},\{\alpha \mapsto \mathit{norm}(\phi)\}\circ\theta)$

(f) $\mathcal{U}(\delta\langle C_1,C_2\rangle,\theta,\mathcal{K}) =$
let $(C_3,\theta_3) = \mathcal{U}(C_1,\theta,\mathcal{K})$ **;** $(C_4,\theta_4) = \mathcal{U}(C_2,\theta,\mathcal{K})$ **in** $(\delta\langle C_3,C_4\rangle,\delta\langle\theta_3,\theta_4\rangle)$

(g) $\mathcal{U}(C_1 \wedge C_2,\theta,\mathcal{K}) =$
let $(C_3,\theta_3) = \mathcal{U}(C_1,\theta,\mathcal{K})$ **;** $(C_4,\theta_4) = \mathcal{U}(\theta_3(C_2),\theta_3,\mathcal{K})$ **in** $(C_3 \wedge C_4,\theta_4\circ\theta_3)$

(h) $\mathcal{U}(C,\theta,\mathcal{K}) = (C,\theta)$

Figure 11 A constraint solving algorithm using pattern matching and guard expressions.

different orderings. The main solver $\mathcal{U}$ has the type $\mathcal{U} : C \times \theta \times 2^{\alpha} \to C \times \theta$, that is, it takes three arguments—the constraint to be solved, the mapping, and the set of type variables for handling idempotent choice eliminations—and returns as two results when constraint solving is successful the residual constraint and the result unifier.

We go briefly over the cases in Figure 11. Case (a) deals with existential constraints of the form $\exists\alpha d.C$, which is satisfiable if C is satisfiable. For this constraint, we first solve C and remove binding information about α and d in the result unifier. Case (b) states that a conditional constraint $\delta \in \phi_1 \Rightarrow \delta{\downarrow}\phi_2$ can be simplified in two ways: (1) if the condition $\delta \in \phi_1$ is satisfied, the constraint is simplified to $\delta{\downarrow}\phi_2$ and (2) if the condition fails, the whole constraint is satisfied. Otherwise, the constraint is deferred for later iterations if the condition contains free dimension or type variables, which may be substituted with concrete dimension names or types.

Case (c), for solving constraints of the form $\delta{\downarrow}\phi_2$, is more interesting. This constraint is processed only when $\theta(\phi_2)$ doesn't contain any free dimensions or type variables (handled by the first guard and its body). Otherwise, if the substituted type doesn't contain any dimensions, then solving fails (handled by the second guard and its body). For example, constraint solving for $\delta{\downarrow}\texttt{Int}$ will fail since `Int` doesn't contain any dimension. Next, if δ is a dimension variable d, then d is mapped to the minimum dimension and the result unifier is updated (handled by the third guard and its body). For example, the solution for $\delta{\downarrow}A\langle\texttt{Int},\texttt{Bool}\rangle$ is $\delta \mapsto A$. Otherwise, if δ doesn't match the minimum dimension, then the constraint is unsatisfiable (handled by the fourth guard and its body). For example, $B{\downarrow}A\langle\texttt{Int},\texttt{Bool}\rangle$ is unsolvable. Finally, if δ is the same as the minimum dimension of ϕ_2, the constraint is trivially satisfiable (last case).

In contrast to previous situations, δ is the same as the minimum dimension, which is handled by the last guard and the corresponding body.

Case (d) deals with type equivalence when both types are variational and contain dimension literals only. In this case, we delegate the work to *vunify*, the variational unification algorithm from [14]. Given two types ϕ_1 and ϕ_2, $\mathit{vunify}(\phi_1, \phi_2)$ returns a pair (θ, r), where θ is the unifier when the unification is successful and r is a boolean value indicating whether the unification problem is solved successfully. The use of *vunify*, together with the fact that type equivalence constraints containing dimension variables are deferred until they are instantiated with dimension literals, makes the constraint solver here simpler than the unification algorithm in VLC [17]. A detailed discussion of the relationship between the constraint solver and VLC can be found in [18].

Case (e) solves a type equivalence constraint between a type variable α and a variational type ϕ, where α doesn't occur in ϕ. In this case, we further decide if $\alpha \in \mathcal{K}$. If so, α is mapped to ϕ. Otherwise, ϕ is simplified with the auxiliary function $\mathit{norm}(\phi)$, and α is mapped to the simplified type. The auxiliary function $\mathit{norm}(\phi)$ reduces idempotent choices and dead alternatives, which are discussed in more detail in [17]. For example, $\mathcal{U}(\alpha \equiv A\langle \texttt{Int}, \texttt{Int}\rangle, \varnothing, \{B\})$ yields $(\texttt{true}, \{\alpha \mapsto \texttt{Int}\})$, and $\mathcal{U}(\alpha \equiv B\langle \texttt{Int}, \texttt{Int}\rangle, \varnothing, \{B\})$ yields $(\texttt{true}, \{\alpha \mapsto B\langle \texttt{Int}, \texttt{Int}\rangle\})$.

Both cases (f) and (g) are straightforward. To solve a variational constraint, we just solve each alternative constraint. To solve the constraint $C_1 \wedge C_2$, we solve both C_1 and C_2. Although the structures for these cases are very similar, there exist two subtle differences. First, C_1 and C_2 are solved independently in (f) while they are solved sequentially in (g). Second, (f) returns a variation of the substitutions for C_1 and C_2 while (g) returns a composition of the substitutions. These differences reflect that the constraints in a variational constraint are independent of each other. Other than previous cases, the constraint is deferred to later iterations, as can be seen in case (h).

$\mathcal{U}$ is sound, complete, and principal, captured in the following theorems.

▸ **Theorem 6** (Soundness of $\mathcal{U}$). *If* $(C_2, \theta_2) = \mathcal{U}(C_1, \theta_1, \mathcal{K})$, *then* $C_2 \Vdash \theta_2(C_1)$ *and* $\theta_2(C_2) = C_2$.

▸ **Theorem 7** (Completeness and principality of $\mathcal{U}$). *Given* (C_1, θ_1), *if* $C_3 \Vdash \theta_3(C_1)$, *then* $(C_2, \theta_2) = \mathcal{U}(C_1, \theta_1, \mathcal{K})$ *and* $\theta_2 \sqsubseteq \theta_3$ *and* $C_3 \Vdash \theta_3(C_2)$, *where* $\theta_2 \sqsubseteq \theta_3$ *if there is some* θ_4 *such that* $\theta_3 = \theta_4 \circ \theta_2$.

Both theorems can be proved by induction on the constraint solving rules in Figure 11.

With $\mathcal{U}'$ and $\mathcal{U}$, we can illustrate how $\mathcal{S}$ solves $\mathcal{C}_1$ from Section 5 through the following sequence of transformations.

$$
\begin{gathered}
(\mathcal{C}_1, \varnothing, \mathcal{K}) \xrightarrow{(e),\textbf{else}} (\{C_2, C_3, C_4\}, \{\alpha \mapsto d\langle \texttt{Int}, \texttt{Bool}\rangle\}, \mathcal{K}) \xrightarrow{(b),1} \\
(\{C_3, C_4; d{\downarrow}\alpha_1\}, \{\alpha \mapsto d\langle \texttt{Int}, \texttt{Bool}\rangle\}, \mathcal{K}) \longrightarrow \\
(\{C_4; d{\downarrow}\alpha_1\}, \{\alpha_1 \mapsto \alpha_2, \alpha \mapsto d\langle \texttt{Int}, \texttt{Bool}\rangle\}, \mathcal{K}) \xrightarrow{(e),\textbf{then}} \\
(\{d{\downarrow}\alpha_1\}, \{\alpha_2 \mapsto A\langle \texttt{Int}, \texttt{Int}\rangle, \alpha_1 \mapsto A\langle \texttt{Int}, \texttt{Int}\rangle, \alpha \mapsto d\langle \texttt{Int}, \texttt{Bool}\rangle\}, \{\alpha_1\}) \xrightarrow{(c),3} (\texttt{true}, \theta_4, \{\alpha_1\}) \\
\theta_4 = \{\alpha_1 \mapsto A\langle \texttt{Int}, \texttt{Int}\rangle, \alpha_2 \mapsto A\langle \texttt{Int}, \texttt{Int}\rangle, d \mapsto A, \alpha \mapsto A\langle \texttt{Int}, \texttt{Bool}\rangle\}
\end{gathered}
$$

The constraint solving process begins with $(\mathcal{C}_1, \varnothing, \mathcal{K})$, where $\mathcal{K} = \mathit{noelim}(\mathcal{C}_1, \{\}) = \{\alpha_1, \alpha_2\}$. Constraint C_1 from $\mathcal{C}_1$ is solved by the **else** branch of case (e) in Figure 11, as indicated by the label "(e),**else**" over the arrow. Next C_2 is solved by the first case of (b), which produces the residual constraint $d{\downarrow}\alpha_1$ that is included in the constraint set. It is separated by a ";" to indicate that the constraint will be solved in the second iteration. Since the case of unifying

two type variables is not included in Figure 11, the arrow for solving C_3 is not labelled. The constraint solving process ends with $(\texttt{true}, \theta_4, \{\alpha_1\})$, where θ_4 is shown at the end of the transformation. This result tells us that the expression *bounded* $A\langle \texttt{2}, \texttt{3} \rangle$ is well typed and has the type $\theta_4(\alpha) = A\langle \texttt{Int}, \texttt{Bool} \rangle$.

On the other hand, $\mathcal{S}(\mathcal{C}_2, \{\})$ fails because the constraint solving process leads to the constraint $d{\downarrow}\texttt{Int}$, which is unsolvable according to the second case of rule (c). Similarly, $\mathcal{S}(\mathcal{C}_3, \{\})$ fails in the second iteration since the constraint $A\langle \texttt{Int}, \texttt{Bool} \rangle \equiv B\langle \texttt{Int}, \texttt{Bool} \rangle$ is unsatisfiable.

In general, $\mathcal{S}$ is sound, complete, and principal since it inherits these properties from $\mathcal{U}$. Together with Theorems 4 and 5, it follows that type inference is sound, complete, and principal.

7 Related Work

In Section 1, we provided references to many applications of variational programming. However, there has not been much work on providing *language support* for variational programming.

The choice calculus (CC) is a formal language for representing *static* variation in trees and other data [23, 50], which is the basis for the choice-based variation representation used in VPC. It provides choices for systematically representing variation, but itself offers no constructs for transforming such representations. There have been a few attempts at extending CC with variational programming features. The compositional choice calculus (CCC) extends CC with functions [51]. It supports generating new variation and a form of variation-preserving computing, but does not provide any mechanisms for eliminating, aggregating, or transforming the variation structure. The main goal of CCC is to unify compositional and annotative approaches to feature implementation, so it also provides a generic mechanism for composing two ASTs. A different extension of CC adds a selection operation to eliminate choices [22]. It focuses mostly on the interaction between selections and a construct for declaring locally-scoped dimensions which we have omitted from VPC, and discusses the impact of the semantics on the modularity of software product line specifications. It does not provide any support for computing with variation.

There have also been several language-based approaches to *compositionally constructing* variational programs with aspects [36], by step-wise refinement [5], or through delta-oriented programming [42]. Each of these provide mechanisms for modularizing changes that may be conditionally applied to a program. As with in-place variation, a substantial amount of work has been done in the software product line community on analyzing and verifying compositionally constructed variational programs. An excellent survey of analyses on both varieties of product line is provided by Thüm et al. [47]. The compositional approach is quite different from our view of in-place variation, so VPC is less applicable in this context. However, one finding of Thüm et al.'s survey is that whole-family analyses are less common over compositional product lines, at least in the literature.

There are a few library-based approaches to variational programming. The concept of variational programming, independent of its applications, was first proposed via a Haskell library [24]. This library requires data types to be explicitly extended by new cases to support variation. Monad instances support convenient variation-preserving computations within a single variational data type, but more advanced use cases—involving multiple variational types or any kind of aggregation or transformation—entail quite some notational overhead to wrap/unwrap variational types and pattern match on variation constructs. This contrasts

with VPC where functions are mapped automatically over arbitrary variational structures. TypeChef [29] is a system developed for parsing and type checking `#ifdef`-annotated source code. To do this, it provides a library of data structures and operations to support variational programming. As a library, it suffers similar inconveniences and limitations as [24]; however, it has been successfully employed in a number projects for analyzing software product lines [37, 38, 33, 30, 34]. Variational data structures and idioms to support variational programming are described in [52], most of which are adapted from the above works.

Work on algebraic effects and effect handlers has frequently used a choice as an example of a non-deterministic effect [4, 8, 27, 41]. These choice effects differ from VPC choices in two key ways: (1) Each choice effect is independent—there is no concept analogous to VPC's dimensions to synchronize selections across choices. (2) The evaluation of an expression with choice effects yields an unstructured set of variants, rather than a structured choice that clarifies the relationship between a sequence of selections and the variant it yields, as in VPC. A notable exception is the "selection functional" effect presented in [4], which essentially implements dimensioned choices in the *Eff* programming language. Both choice effects and selection functionals are more restrictive than VPC choices since they do not permit the alternatives of the choice, or the produced variants, to be values of different types. Additionally, the control flow enforced by these encodings of choices rules out several ways to optimize variation-preserving computations. Since computations for different alternatives are performed in different continuations, we lose the opportunity to perform choice reduction to proactively eliminate unreachable alternatives [17], or to join converged execution paths early. Finally, encoding choices as effects loses the ability to explicitly reflect on the structure of a variational expression and perform transformations. This ability is often needed for variational analyses, for example, the ability to commute selections with computations is a critical transformation for analyzing SPLs [29, 28, 46, 2, 17, 16]. All of the limitations of choice effects and selection functionals relative to VPC can likely be overcome through more clever effect encodings. However, this would essentially amount to embedding VPC into a host language with a rather complicated semantics, complicating our understanding of effective variational programming.

Variation-preserving computations can also be encoded as computations in a reader monad, where the environment identifies a single variant (for example, a predicate on dimensions) and choices are encoded as conditional statements querying this environment. Some of the limitations of this encoding echo the above: It does not support alternatives of different types, and it doesn't support optimizations or transformations since we cannot reflect on the structure of variation in the program. The reader-based encoding has the additional drawback of describing the variational results of a computation intensionally, rather than extensionally. That is, the result is a function that can be used to obtain different variants by passing in different environments, rather than an explicit choice structure as obtained from VPC. This makes it prohibitively expensive to enumerate the variants of a result since we must try all possible selections of the dimensions, rather than just iterating over the variants at the leaves of a choice expression.

There are, of course, many applications that do some form of variation programming, but without any specific programming support. The idea behind *multi-execution* is to run small number of program alternatives in parallel; it has been applied to identify security problems [21, 20, 9, 32], configuration bugs [43], and update inconsistencies [49, 26, 35] in programs. Most of these applications synchronize the different executions externally, but some approaches do exploit similarities between programs run in parallel [49, 45]. Most of these approaches are limited to running two programs at a time and would not scale to number of variants encountered in variational programming.

8 Conclusions

Variational programming supports a wide variety of applications that must somehow deal with variation in programs or data, but computing with and transforming variation representations in general-purpose programming languages is effort intensive and error prone. In this paper, we have presented VPC, a core calculus that supports the implementation of a variational functional programming language that provides direct support for variational programming. This language directly supports the implementation of high-level functions for aggregating and transforming variation, as illustrated in Section 2. Additionally, it enables a very simple implementation of a variational type unification algorithm, which is quite complicated without built-in support for variation [17]. As future work we intend to implement other pre-existing variational programming applications using VPC.

We have defined a small-step operational semantics for VPC, and demonstrated that it offers flexibility in its reduction strategy. In existing variational programming applications, much effort is often needed to demonstrate that computations and variation commute properly, but this is built into the semantics of VPC. We presented a type system that relates variational types with VPC expressions. The type system supports dimension polymorphism through a restricted form of dependent types. Finally, we described a type inference algorithm for automatically inferring most general types for VPC expressions. At the core of this algorithm is a new variational constraint solving algorithm.

Acknowledgments. We would like to thank the anonymous reviewers for exceptionally detailed comments that improved the quality of this paper.

References

1 S. Apel, D. Beyer, K. Friedberger, F. Raimondi, and A. von Rhein. Domain types: Abstract-domain selection based on variable usage. In *HVC*, LNCS 8244, pages 262–278, 2013.
2 S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type safety for feature-oriented product lines. *JASE*, 17(3):251–300, 2010.
3 T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *POPL*, pages 165–178, 2012.
4 Andrej B. and Matija P. Programming with algebraic effects and handlers. *CoRR*, abs/1203.1539, 2012.
5 D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *TSE*, 30(6):355–371, 2004.
6 E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. SPL^{LIFT}: Statically analyzing software product lines in minutes instead of years. In *PLDI*, pages 355–364, 2013.
7 C. Brabrand, M. Ribeiro, T. Tolêdo, and Paulo B. Intraprocedural dataflow analysis for software product lines. In *AOSD*, pages 13–24, 2012.
8 E. Brady. Programming and reasoning with algebraic effects and dependent types. In *ICFP*, pages 133–144, 2013.
9 R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. P. Sistla. Preventing information leaks through shadow executions. In *ACSAC*, pages 322–331, 2008.
10 S. Chen and M. Erwig. Counter-factual typing for debugging type errors. In *POPL*, pages 583–594, 2014.
11 S. Chen and M. Erwig. Early detection of type errors in C++ templates. In *PEPM*, pages 133–144, 2014.
12 S. Chen and M. Erwig. Guided type debugging. In *FLOPS*, LNCS 8475, pages 35–51. 2014.

13 S. Chen and M. Erwig. Type-based parametric analysis of program families. In *ICFP*, pages 39–51, 2014.

14 S. Chen and M. Erwig. Principal type inference for GADTs. In *POPL*, pages 416–428, 2016.

15 S. Chen, M. Erwig, and K. Smeltzer. Let's hear both sides: On combining type-error reporting tools. In *VL/HCC*, pages 145–152, 2014.

16 S. Chen, M. Erwig, and E. Walkingshaw. An error-tolerant type system for variational lambda calculus. In *ICFP*, pages 29–40, 2012.

17 S. Chen, M. Erwig, and E. Walkingshaw. Extending type inference to variational programs. *TOPLAS*, 36(1):1:1–1:54, 2014.

18 S. Chen, M. Erwig, and E. Walkingshaw. A calculus for variational programming. Technical report, University of Louisiana at Lafayette and Oregon State University, 2016. URL: `http://shengchen1.bitbucket.org/ws/TechReport/vpc.pdf`.

19 M. d'Amorim, S. Lauterburg, and D. Marinov. Delta execution for efficient state-space exploration of object-oriented programs. *TSE*, 34(5):597–613, 2008.

20 W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: A web browser with flexible and precise information flow control. In *CCS*, pages 748–759, 2012.

21 D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *S&P*, pages 109–124, 2010.

22 M. Erwig, K. Ostermann, T. Rendel, and E. Walkingshaw. Adding configuration to the choice calculus. In *VAMOS*, pages 13:1–13:8, 2013.

23 M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *TOSEM*, 21(1):6:1–6:27, 2011.

24 M. Erwig and E. Walkingshaw. Variation programming with the choice calculus. In *GTTSE*, LNCS 7680, pages 55–100, 2013.

25 R. Harper. *Practical Foundations for Programming Languages (2ed)*. Cambridge University Press, New York, NY, USA, 2016.

26 P. Hosek and C. Cadar. Safe software updates via multi-version execution. In *ICSE*, pages 612–621, 2013.

27 O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *ICFP*, pages 145–158, 2013.

28 C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *TOSEM*, 21(3):14:1–14:39, 2012.

29 C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *OOPSLA*, pages 805–824, 2011.

30 C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward variability-aware testing. In *FOSD*, pages 1–8, 2012.

31 C. H. P. Kim, S. Khurshid, and D. Batory. Shared execution for efficiently testing product lines. In *ISSRE*, pages 221–230, 2012.

32 C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *S&P*, pages 443–457, 2012.

33 J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer. Morpheus: Variability-aware refactoring in the wild. In *ICSE*, pages 380–391, 2015.

34 J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *ESEC/FSE*, pages 81–91, 2013.

35 M. Maurer and D. Brumley. TACHYON: Tandem execution for efficient live patch testing. In *USENIX Security Symposium*, pages 43–43, 2012.

36 M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. *SEN*, 29(6):127–136, 2004.

37 S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *ICSE*, pages 140–151, 2014.

38 S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Where do configuration constraints stem from? An extraction approach and an empirical study. *TSE*, 2015.

39 H. Nguyen, C. Kästner, and T. N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *ICSE*, pages 907–918, 2014.

40 M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

41 G. Plotkin and J. Power. Adequacy for algebraic effects. In *FoSSaCS*, pages 1–24, 2001.

42 I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Software Product Lines: Going Beyond*, pages 77–91. 2010.

43 Y. Su, M. Attariyan, and J. Flinn. AutoBash: Improving configuration management with operating system causality analysis. In *SOSP*, pages 237–250, 2007.

44 M. Sulzmann. A general type inference framework for hindley/milner style systems. In *FLOPS*, pages 248–263, 2001.

45 W. N. Sumner, T. Bao, X. Zhang, and S. Prabhakar. Coalescing executions for fast uncertainty analysis. In *ICSE*, pages 581–590, 2011.

46 S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *GPCE*, pages 95–104, 2007.

47 T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *CSUR*, 47(1):6, 2014.

48 E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, pages 530–541, 2014.

49 J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. In *ASPLOS*, pages 193–204, 2009.

50 E. Walkingshaw. *The choice calculus: A formal language of variation.* PhD thesis, Oregon State University, 2013.

51 E. Walkingshaw and M. Erwig. A calculus for modeling and implementing variation. In *GPCE*, pages 132–140, 2012.

52 E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. Variational data structures: Exploring trade-offs in computing with variability. In *Onward!*, pages 213–226, 2014.

53 E. Walkingshaw and K. Ostermann. Projectional editing of variational software. In *GPCE*, pages 29–38, 2014.

Interprocedural Type Specialization of JavaScript Programs Without Type Analysis*

Maxime Chevalier-Boisvert[1] and Marc Feeley[2]

1 **DIRO, Université de Montréal**
Montreal, Quebec, Canada
chevalma@iro.umontreal.ca

2 **DIRO, Université de Montréal**
Montreal, Quebec, Canada
feeley@iro.umontreal.ca

Abstract

Previous work proposed lazy basic block versioning, a technique for just-in-time compilation of dynamic languages which we believe represents an interesting point in the design space. Basic block versioning is simple to implement, simple enough that a single developer can build a complete just-in-time compiler for JavaScript in a year, yet it performs surprisingly well as it propagates context-sensitive type information to generate type-specialized code on the fly.

In this paper, we demonstrate that lazy basic block versioning can be extended in simple ways to propagate type information across function call boundaries. This gives some of the benefits of whole-program analysis, or a tracing compiler, without having to implement the machinery for either. We have implemented this proposal in the Higgs JavaScript virtual machine and report on the empirical evaluation of this system on a set of industry standard benchmarks. The approach eliminates 94.3% of dynamic type tests on average, which we show is more than what is achievable with any static whole-program type analysis.

1998 ACM Subject Classification D.3.4 Programming Languages: Processors—compilers, optimization, code generation, run-time environments

Keywords and phrases Just-In-Time Compilation, Dynamic Language, Optimization, Object Oriented, JavaScript

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.7

1 Introduction

A production compiler for a widely used dynamic language such as JavaScript is an intricate piece of software, usually the outcome of 10 to 100 developer-years of effort. The architecture of such a compiler is one of the first design decisions made during development. This decision is rarely revisited, as architectural changes tend to be disruptive. In previous work, Chevalier-Boisvert and Feeley argued for an architecture based on the concept of lazy Basic Block Versioning (BBV) [14]. They claimed that the technique hits a sweet spot in the tradeoff between implementation complexity and performance of the generated code. As evidence they designed and implemented Higgs, a JavaScript virtual machine and Just-In-Time (JIT) compiler which has performance competitive with other research virtual machines and can sometimes match the performance of production systems such as V8. Notably, the

* This work was supported, in part, by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Mozilla Corporation.

Higgs compiler took about a year of development time. The reduced development time is particularly important for languages that are maintained by small teams of volunteers. Lazy BBV occupies a point in the design space of JIT compilers that is between method-based compilers and tracing JITs such as Mozilla's TraceMonkey [17], and run-time specialization of Oracle's Truffle [36]. The simplicity of BBV is one of its main advantages. It does not require additional infrastructure such as a static analyzer to approximate program facts, or an interpreter to record traces.

BBV is a simple and elegant compilation technique to optimize dynamically typed programs on the fly. The technique uses dynamic type tests which are part of the implicit semantics of primitive operators in dynamically typed languages to capture and propagate type information. Type-specialized versions of individual basic blocks are lazily compiled based on the types encountered during the execution of programs. The technique, as described in [14], is limited to optimizing type checks on local variables within a single function. The compiler has no information on the types of arguments, return values, or object properties, and is thus unable to eliminate some redundant dynamic type checks.

This paper extends basic block versioning with the ability to propagate type information across function call boundaries and to specialize code based on the type of object properties. In the framework of basic block versioning, these extensions are easy to implement and seem to work rather well. This paper makes the following specific contributions:

1. The combination of BBV with a *typed object shape* mechanism which encodes property type information including *method identity*, enabling the compiler to know the identity of callees at call sites (Section 4.1).
2. The extension of BBV with *specialized function entry points*, which makes it possible to pass argument types from callers to callees. This is done efficiently, without dynamic dispatch, using method identity information provided by typed object shapes (Section 4.2).
3. A speculative technique for *call continuation specialization*, which enables type information about return values to be passed from callees back to callers, without dynamic overhead (Section 4.3).

To validate our claims we implemented these contributions in the Higgs JavaScript compiler and evaluated its performance on industry standard benchmarks (Section 5).

A word about evaluation is in order. We considered implementing our ideas within an existing JavaScript compiler, but quickly realized that the architectural changes required were beyond our resources. Thus we picked Higgs as a vehicle for our experiments. This choice comes at a cost; comparing performance of a research prototype to a production system is tricky. A production system has a mature garbage collector, highly tuned libraries, and performs a massive number of optimizations (many, but not all, of which are orthogonal to this work). A research prototype is likely to not have any of those. It is thus not surprising that Higgs runs roughly half as fast as V8. This may be a sign that our approach is inherently limited, or that we simply lack the resources of major corporations. Cognizant of the inherent limitations of empirical evaluations, we have chosen the following approach. We measure the improvement of the techniques presented in this paper by the number of type tests we are able to eliminate and the performance impact over the previous version of the Higgs compiler. This gives us a metric of progress. We compare our implementation with two relevant systems, one is the TraceMonkey tracing compiler. The reason for this comparison is that basic block versioning has been compared by others to tracing compilation. It is thus interesting to see how the two perform on the same benchmarks. Then we choose Truffle/JS as an example of a research prototype, albeit one implemented by a large team of industrial

researchers. For completeness we include, in Appendix A, performance results comparing Higgs to leading commercial JavaScript implementations.

2 Influences and Related Work

The literature on just-in-time compilation is rich with, by now, decades of work. The work presented here was influenced by many results obtained in the Self project and should be contrasted to work on type analysis and dynamic compilation of dynamic languages.

Shapes. The notion of describing objects with shapes can be traced back to the Self programming language [11, 22], where so-called maps group objects cloned from the same prototype. Like shapes, maps reduce memory usage and stored metadata relating to properties (though not type information). Today, commercial JavaScript implementations such as V8, SpiderMonkey, Nitro and Truffle/JS have all adopted this idea. Each object contains a pointer to its shape, which describes the layout of the object and property attribute metadata. Truffle introduced the notion of specializing shapes based on property types to the literature [35]. This paper builds on that idea and demonstrates how to effectively integrate such a model with basic block versioning.

Splitting. Basic block versioning bears resemblance to Self's *iterative type analysis* and *extended message splitting* [13] which combines static analysis with a transformation that compiles multiple versions of loops and duplicates control flow paths to eliminate type tests. The analysis works in an iterative fashion, transforming the control flow graph of a function while performing a type analysis. It integrates a mechanism to generate new versions of loops when needed, and a message splitting algorithm to try and minimize type information lost through control flow merges. One key disadvantage is that statically cloning code requires being conservative, generating potentially more code than necessary, as it is impossible to statically determine exactly which control flow paths will be taken at run time, and this must be overapproximated. The approach also has roots in Agesen's cartesian product algorithm [2] which avoids the loss of type information at control-flow merges by representing program state with sets of vectors of concrete types.

Analysis. There have been multiple efforts to devise type analyses for dynamic languages. Rapid Atomic Type Analysis [27] is an intraprocedural flow-sensitive analysis that assigns unique types to each variable. Attempts have also been made to define formal semantics for a subset of dynamic languages such as JavaScript [5], Ruby [16] and Python [4], sidestepping some of the complexity of these languages and making them more amenable to traditional type inference techniques. There are also flow-based interprocedural type analyses for JavaScript based on sophisticated type lattices [23, 24, 25]. Such analyses are too time consuming to be used in a just-in-time compiler. Kedlaya, Roesch et al. [26] improved the precision of type analyses by combining them with type feedback and profiling. This shows promise, but does not deal with object shapes and property types. Work has also been done on a flow-sensitive alias analysis for dynamic languages [19], but it is still unclear if the analysis can be used on-line. More recently, Brian Hackett et al. presented an interprocedural hybrid type analysis for JavaScript suitable for use in a just-in-time compiler [21]. While this is an important step forward, it remains vulnerable to imprecise type information polluting analysis results. Basic block versioning can help improve on the results of such an analysis by hoisting tests out of loops and generating multiple optimized code paths where appropriate.

Tracing. Trace compilation, introduced by Dynamo [6] and later applied to just-in-time compilation in HotpathVM [18], aims to record sequences of instructions executed inside hot loops. Such sequences make optimization simpler. Type information is accumulated along traces and used to specialize code and remove type tests [17], overflow checks [34] or unnecessary allocations [8]. Basic block versioning resembles tracing in that context updating works on essentially linear code fragments and code is optimized similarly to what may be done in a tracing compiler. Code is also compiled lazily, as needed, without compiling whole functions at once. Trace compilation [9] and meta-tracing are an active area of research [10]. The simplicity of basic block versioning is one of its main advantages. It does not require external infrastructure such as an interpreter to record traces. Trace compiler implementations must deal with corner cases that do not appear with basic block versioning. With trace compilation, there is the potential for trace explosion if there are a large number of control flow paths going through a loop [7]. It is also not obvious how many times a loop should be recorded or unrolled to maximize the elimination of type checks. This problem is solved with basic block versioning since versioning is driven by type information and there is a natural bound to the number of versions that comes from the finite number of types in the system. Trace compilers must implement parameterizable policies and mechanisms to deal with recursion, nested loops and potentially very long traces that do not fit in instruction caches.

Customization. Customization is another technique developed to optimize Self programs [12]. It compiles multiple copies of methods specialized on the receiver object type. Similarly, *type-directed cloning* [28] clones methods based on argument types, producing more specialized code using richer type information. The work of Chevalier-Boisvert et al. on just-in-time specialization for MATLAB [15] and similar work done for the MaJIC MATLAB compiler [3] tries to capture argument types to dynamically compile optimized versions of whole functions. All of these techniques are forms of type-driven code duplication aimed at extracting type information. Basic block versioning operates at a lower level of granularity, allowing it to find optimization opportunities inside of method bodies by duplicating code paths. There are similarities between the Psyco JIT specialization work and our own. The Psyco prototype for Python [31] interleaves execution and JIT compilation to gather run time information about values. It then specializes code on the fly based on types and values. It also incorporates a scheme where functions can have multiple entry points. We extend upon this work by combining a similar approach, that of basic block versioning, with typed shapes and a mechanism for propagating return types from callees to callers with low overhead. The *tracelet-based* approach used by Facebook's *HHVM* for PHP [1] bears similarities to our own. HHVM compiles small code regions (tracelets) which are single-entry multiple-exit basic blocks. Each tracelet is type-specialized based on variable types observed at compilation time. Guards are inserted at the entry of tracelets to verify at run time that the types observed are still valid for all future executions. High-level instructions in tracelets are specialized based on the guarded types. If these guards fail, new versions of tracelets are compiled based on different type information and chained to the failing guards. One difference with our work is that HHVM uses an ahead-of-time type analysis pass. Another difference is that with the approach described in [1], each tracelet re-checks the types of its inputs, whereas BBV propagates known types to successor blocks and doesn't usually need to re-check the types of local variables. Finally, HHVM falls back on an interpreter when too many tracelet versions are generated. Higgs falls back to generic basic block versions which do not make type assumptions but are still compiled. Beyond type specialization, recent work by Costa et al. on *just-in-time value specialization* has shown that specializing JavaScript functions based

on specific argument values can lead to performance improvements [33], as many functions are always called with the same arguments.

3 Background

The work presented in this paper is implemented in a research virtual machine for JavaScript (ECMAScript 5) known as Higgs [1]. The Higgs virtual machine includes a just-in-time compiler built around lazy basic block versioning. This compiler is intended to be lightweight with a simple implementation. Code generation and type specialization are performed in a single pass. Register allocation is done using a greedy allocator. The runtime and standard libraries are self-hosted, written in an extended dialect of JavaScript with low-level primitives. These low-level primitives are special instructions which allow expressing type tests, pointer manipulation, as well as integer and floating point machine instructions in the source language.

3.1 Value Types and Type Tests

Higgs segregates values into categories based on type tags [20]. These type tags form a simple type system that is used for versioning. The types are mostly straightforward and correspond closely to values manipulated by JavaScript programs. The one exception is the `unknown` type tag that is used by the compiler to indicate that no information is available for the corresponding value.

`int32`	signed 32-bit integers
`float64`	64-bit floating point numbers
`undef`	the `undefined` value
`null`	the `null` value
`bool`	`true` and `false` boolean values
`string`	strings
`array`	arrays
`closure`	function objects
`object`	Plain JS objects
`unknown`	type unknown

JS is a dynamically typed and late-bound programming language. There are no static type annotations, and the types of variables may change during the execution of a program. As such, there are many implicit type checks hiding in even the simplest JS programs. Figure 1 shows an iterative function which illustrates this. The `sum` function contains three primitive operators: a comparison, a decrementation and an addition. Each of these operators implicitly checks the types of its operands as part of its semantics.

In all, there are four implicit type checks hiding in the `sum` function:

1. The `>` operator checks the type of `n` before comparing it against the integer zero.
2. The type of `s` is checked before computing `s += i`
3. The type of `i` is also checked before computing `s += i`
4. The decrementation operator checks the type of `i` before computing `--i`

[1] https://github.com/higgsjs

```
function sum(n) {
    var s = 0;
    for (var i = n; i > 0; --i)
        s += i;
    return s;
}

sum(500);
```

Figure 1 Iterative JS `sum` function.

```
A: s = 0, i = 0
   if not is_int32(n) goto stub1
B: if not gt_int32(n,0) goto I  // if not (n > 0)

C: if not is_int32(s) goto stub2
D: if not is_int32(i) goto stub3
E: s = add_int32(s,i)
   if overflow goto stub4

F: if not is_int32(i) goto stub5
G: i = sub_int32(i,1)
   if overflow goto stub6
H: goto B

I: return s
```

Figure 2 Control-flow graph of the `sum` function before BBV.

A naive JS implementation performs these type checks every time an operator is evaluated. In Higgs, this is done using primitive instructions which can test the type tags of values. Figure 2 illustrates the primitive operations and implicit type tag checks executed by Higgs with basic block versioning disabled when `sum(500)` is evaluated. When computing `sum(500)`, only small integer (`int32`) values are used, and so, much of these type checks are redundant.

The `is_int32` primitives act as guards which verify that the type tag associated with a given variable is `int32` before executing a machine instruction specific to integer values. Should any of these tests fail, execution will flow to a stub that generates new machine code to handle non-integer values. The overflow test primitives serve to verify that an integer overflow did not occur, and handle such an occurrence otherwise.

3.2 Lazy Basic Block Versioning

Basic block versioning is a just-in-time code generation technique originally applied to JavaScript by Chevalier-Boisvert & Feeley [14], and adapted to Scheme by Saleil & Feeley [32]. The technique bears similarities to HHVM's tracelet-based compilation approach and Psyco's just-in-time code specialization system [31].

BBV works at the level of individual basic blocks. We define a basic block as a single-entry single-exit sequence of instructions. Basic blocks end with one branching instruction which jumps to other basic blocks. In Higgs, basic blocks are usually short, sometimes just one instruction in our Intermediate Representation (IR), due to the large number of type tests, each of which is treated as a branching instruction which terminates the current basic block.

The BBV engine interleaves compilation and execution. It generates machine code for basic blocks lazily, instantiating them into one or more versions, each type-specialized based on accumulated type information. BBV propagates type information by maintaining a context for each block version which stores known type information about live variables.

```
A: s = 0, i = 0
   if not is_int32(n) goto stub1
B: if not gt_int32(n,0) goto I

// s,i,n are known to be int32
C: //is_int32(s) check eliminated
D: //is_int32(i) check eliminated
E: s = add_int32(s,i)
   if overflow goto stub2

// s,i,n are known to be int32
F: //is_int32(i) check eliminated
G: i = sub_int32(i,1)
   if overflow goto stub3
H: goto B

I: return s
```

Figure 3 Control-flow graph of the `sum` function after BBV.

This context is updated as block versions are compiled.

Type tag tests are used to capture type information and enrich the versioning context. We know, for instance, that if we branch on the "true" side of an `is_int32(n)` test, then `n` must have tag `int32` in the successor block. This fact is exploited by instantiating a specialized version of the successor block based on the knowledge that `n` is `int32`. Because BBV uses lazy code generation, it never generates block versions for types that do not occur at run time. It achieves this by delaying the compilation of conditional branch targets using machine code stubs.

Using BBV, three of the four implicit type checks in the `sum` function from Figure 1 are eliminated. The resulting optimized control flow graph is shown in Figure 3. A single type test remains: the type of `n` is tested when entering the function. When first executing the `sum(500)` call, Higgs takes the following steps to compile and optimize the `sum` function:

- The `sum` function is entered, block `A` is executed. The `s` and `i` variables are initialized to `0`. The context is updated to indicate both `s` and `i` have type tag `int32`. The type of `n` is unknown. The `is_int32(n)` branch is made to point to machine code stubs and execution is resumed.
- Execution resumes. The `is_int32(n)` check evaluates to "true". A stub for block `B` is hit. This stub calls back into the compiler.
- Compilation resumes, and a version of block `B` with `n` known to be `int32` is generated. Stubs are generated for the `gt_int32(n,0)` branch targets.
- Execution resumes. A stub of block `C` is hit.
- Compilation resumes. A version of block `C` with `n` known to be `int32` is produced. The variables `s` and `i` are already known to be `int32`, hence the type tag checks in `C` and `D` can be evaluated at compilation time and eliminated. A stub is produced for the integer overflow check.
- Execution resumes. No overflow occurs, a stub for block `F` is hit.
- Compilation resumes. A version of block `F` with `s`, `i` and `n` as `int32` is compiled. The type check in `F` is evaluated at compilation time and eliminated. Stubs for the overflow branch in `G` are produced.
- Execution resumes. No overflow occurs, a stub of block `H` is hit.
- Compilation resumes. Block `H` produces a jump to the version of `B` that was already generated, where `s`, `i` and `n` are all known to be `int32`.

```
function sumList(lst) {
    if (lst == null)
        return 0
    return lst.val + sumList(lst.next)
}

function makeList(len) {
    if (len == 0)
        return null
    return { val: len, next: makeList(len-1) }
}

var lst = makeList(100)
if (sumList(lst) != 5050)
    throw Error('incorrect␣sum')
```

Figure 4 JS function to recursively sum the values stored in a linked list.

- Execution resumes and continues until the `gt_int32(n,0)` test in block `B` fails. Note that no more type checks are executed.
- The loop test fails. A stub for block I is hit. Block `I` is compiled.
- Execution resumes at block `I`, the `sum` function returns to the caller.

Because of its JIT nature, BBV has at least two powerful advantages over traditional static type analyses. The first is that BBV considers only the parts of the control flow graph that get executed, and it knows precisely which they are, as machine code is only generated for basic blocks which are executed. The second is that code paths can often be duplicated and specialized based on different type combinations, making it possible to avoid the loss of precision caused by control flow merges in traditional type analyses.

3.3 Motivating Example

The example in Figure 1 is one for which plain intraprocedural BBV works particularly well. In this section, we will provide a motivating example for our work which highlights the limitations of the unextended BBV approach described in [14]. We will then show how we have extended BBV to remove these limitations.

Figure 4 shows the `sumList` function for recursively traversing a linked list and computing the sum of numerical values stored in each node. While this small program may appear simplistic, there is much semantic complexity hidden behind the scenes. A correct but naive implementation of this function contains six implicit dynamic type tag tests, which must be eliminated to maximize performance:

1. The tag of the `lst` argument is checked when comparing it against `null`.
2. The tag of `lst` is re-checked before reading the `lst.val` property.
3. The tag of `lst` is checked a third time before reading the `lst.next` property.
4. The `sumList` function is a mutable global variable. Before calling it, there is an implicit check to make sure that this is in fact a closure.
5. The tag of `lst.val` is checked before computing `lst.val + sumList(lst.next)`.
6. The tag of `sumList(lst.next)` is also checked, because functions calls can return values of any type.

The BBV algorithm described in [14] is limited to an intraprocedural scope, that is, it deals with local variable types only. It cannot pass type information between callers and callees. It also assumes that all object properties (including global variables, which are

```
A: if is_null(lst) goto I

B: if not is_object(lst) goto stub1
C: val = read_prop(lst, 'val')
   if not is_object(lst) goto stub2

D: next = read_prop(lst, 'next')
   sumfn = read_prop(globalObj, 'sumList')
   if not is_closure(sumfn) goto stub3

E: t1 = sumfn(next)
   if not is_int32(val) goto stub4

F: if not is_int32(t1) goto stub5

G: t2 = add_int32(val, t1)
   if overflow goto stub6

H: return t2

I: return 0
```

Figure 5 Implicit type checks in the `sumList` function.

properties of the global object) have unknown type. As such, the unextended BBV algorithm is ill-equipped to optimize the `sumList` function, or object-oriented JS code in general.

The implicit tests executed by a version of Higgs without BBV are shown in Figure 5.

Once the type tag of the `lst` parameter has been tested and found to be `object`, intraprocedural BBV can eliminate the second `is_object` test. Unfortunately, it cannot eliminate any of the other type tag tests. Since nothing is known about object property types, the type tags of the `val` and `next` properties must be tested for each call. The type tag of `sumList` is also tested before every call. Lastly, the return type of the `sumList` call is checked after each call. Clearly, most of these checks are provably redundant, and it should be feasible to eliminate them. The next sections will explain the ways in which we have extended BBV to give it the necessary capabilities.

4 Interprocedural Basic Block Versioning

This section describes the three extensions to basic block versioning that allow us to propagate type information across procedure calls.

4.1 Typed Object Shapes

BBV, as presented in [14], deals with function parameter and local variable types only. It has no mechanism for attaching types to object properties. This is particularly problematic because, in JS, functions are typically stored in objects. This includes object methods and also global functions (JS stores global functions as properties of the *global object*). We would like to attach type tags to object properties, global variables included.

4.1.1 Object Shapes and Shape Tests

Currently, all commercial JS engines have a notion of object shapes, which is similar to the notion of property maps invented for the Self VM. That is, any given object contains a pointer to a shape descriptor providing its memory layout: the properties it contains, the

```
// Linked list node shape
S: { val: slot 0, next: slot 1 }

// Global object shape
G: {
    ...,
    Error: slot 1,
    ...,
    makeList: slot 30,
    sumList: slot 33,
    lst: slot 34
}
```

Figure 6 Linked list node and global object shapes.

```
A: if is_null(lst) goto I

B: if not is_object(lst) goto stub1
C: if not is_shape(lst, S) call updatePIC  // PIC 1
   val = read_slot(lst, 0)                 // PIC 1
   if not is_object(lst) goto stub2

D: if not is_shape(lst, S) call updatePIC  // PIC 2
   next = read_slot(lst, 1)                // PIC 2
   if not is_shape(globalObj, G) call updatePIC // PIC 3
   sumfn = read_slot(globalObj, 33)            // PIC 3
   if not is_closure(sumfn) goto stub3

E: t1 = sumfn(next)
   if not is_int32(val) goto stub4

F: if not is_int32(t1) goto stub5

G: t2 = add_int32(val, t1)
   if not overflow goto stub6

H: return t2

I: return 0
```

Figure 7 Primitive operations in `sumList` executed by an unextended version of Higgs.

property slot index (memory offset) each property is stored at, as well as attribute flags (i.e. writable, enumerable, etc.). For instance, linked list nodes and the global object in the example from Figure 4 have shapes `S` and `G`, shown in Figure 6.

Traversing shape data structures on each object property access would be prohibitively expensive. As such, Higgs and all modern JS engines optimize property accesses using Polymorphic Inline Caches (PICs) [22]. PICs are lazily updated sequences of inlined machine instructions which implement property reads and writes. Typically, a cascade of conditional branch instructions establish the shape of an object in order to determine the memory offset at which the property to be read or written is stored. A specialized machine instruction is then executed which accesses the property at the correct offset. PICs are extended as needed to handle previously unseen object shapes.

In the `sumList` function, there are three property reads, and therefore three PICs. Linked list nodes and the global object only have one possible shape, and so there is only one shape test inside each PIC. The primitive operations and dynamic tests executed by an unextended implementation of Higgs which uses PICs are illustrated in Figure 7.

```
// Linked list node shape
S1: { val: (slot 0, int32), next: (slot 1, null) }
S2: { val: (slot 0, int32), next: (slot 1, object) }

// Global object shape
// Closures have method identity information
G: {
    ...,
    Error: (slot 1, closure/Error),
    ...,
    makeList: (slot 30, closure/makeList),
    sumList: (slot 33, closure/sumList),
    lst: (slot 34, object)
}
```

Figure 8 Typed object shapes encode property type information.

4.1.2 Extending Shapes with Types

Work done on the Truffle Object Model (OSM) [35] describes how object shapes can be straightforwardly extended to also encode type tags for object properties. Property writes are guarded to update object shapes when a property type changes. Property reads establish the shape of objects in order to know the memory offset at which to read properties. When object shapes also encode the type tags of properties, establishing the shape of an object tells us not only where to read the property, but also what type tag this property has. Hence, the cost of guarding property writes is easily offset, because typical JS programs have many more property reads than property writes. A small overhead is paid to guard property writes, and in exchange, type checks after property reads are effectively eliminated.

We extend upon the original BBV work with a *typed object shape* system inspired by the Truffle OSM. This model is a natural fit for the BBV algorithm. Our extended BBV algorithm not only propagates known type tags associated with values, but also object shapes. The shape tests which are normally part of PICs allow our JIT compiler to establish and propagate the shape of an object in the same way that type tag tests enabled BBV to extract and propagate the type tags of values. Once the shape associated with an object is known to the BBV engine, then, by extension, the types of all properties read from that object are also known.

In order to enable interprocedural type propagation, it is useful to know which function is being called for as many call sites as possible, both for calls to global functions and method calls. As such, we have gone one step further than the Truffle OSM, and attached not only type tags to object shapes, but also *method identity information*. That is, for properties which have the closure type tag, shapes encode a pointer to the IR node corresponding to the function the property is a closure of. This enables us to know the identity of callees at code generation time for the large majority of call sites.

With typed shapes, linked list nodes from the `sumList` have two possible shapes, one where the `next` property is `null`, and one where it is an object. The global object encodes not only the offsets of global variables, but also the identity of global functions. This is illustrated in Figure 8.

In order to allow BBV to take advantage of typed shape information, we break up PICs into their component parts. PICs, which were previously monolithic sequences of inlined machine instructions, are now exposed in our compiler IR as separate shape test and memory access instructions. The result is that the regular BBV mechanisms can be leveraged to extract shape information from shape tests and propagate it. Propagating shape information (and the associated property types), allows us to optimize the `sumList` function as shown in Figure 9.

```
A: if is_null(lst) goto I:

B: if not is_object(lst) goto stub1
C: if not is_shape(lst, S1) goto C2
   val = read_slot(lst, 0) // val is known to be int32
   next = read_slot(lst, 1) // next is known to be null

D: if not is_shape(globalObj, G) goto stub2
   sumfn = read_slot(globalObj, 33) // sumfn is known to be a closure

E: t1 = sumfn(next)
   if not is_int32(t1) goto stub3

G: t2 = add_int32(val, t1)
   if overflow goto stub4

H: return t2

I: return 0

C2: if not is_shape(lst, S2) goto stub5
    val = read_slot(lst, 0) // val is known to be int32
    next = read_slot(lst, 1) // next is known to be object

D2: if not is_shape(globalObj, G) goto stub6
    sumfn = read_slot(globalObj, 33) // sumfn is known to be a closure

E2: t1 = sumfn(next)
    if not is_int32(t1) goto stub7

G2: t2 = add_int32(val, t1)
    if overflow goto stub8

H2: return t2
```

Figure 9 The `sumList` function optimized with typed shapes.

Two separate code paths are generated inside the `sumList` function, one for each of the two possible shapes of the linked list nodes. More code is generated, but on any given code path, at most three type tag tests are executed instead of five. Since linked list nodes now have two possible shapes, we may test the shape of linked list nodes twice instead of just once when reading the `lst.val` property. However, because we no longer employ monolithic inline caches, this shape is propagated from the property read of `lst.val` to that of `lst.next`. Hence, as a result, we actually perform less dynamic shape tests on average.

4.2 Entry Point Versioning

Procedure cloning has been shown to be a viable optimization technique, both in ahead of time and JIT compilation contexts. By specializing function bodies based on argument types at call sites, it becomes possible to infer the types of a large proportion of local variables, allowing effective elimination of type checks.

Our first extension to BBV is to allow functions to have multiple type-specialized entry points. That is, when the identity of a callee at a given call site is known at compilation time, the JIT compiler requests a specialized version of the entry point block for the callee. This specialized entry point assumes the argument types known at the call site. Type information is thus propagated from the caller to the callee.

Inside the callee, BBV proceeds as described in [14], deducing local variable types and eliminating redundant type checks. Our approach places a hard limit on the number of

versions that may be created for a given basic block, and so automatically limits the number of entry points that may be created for any given function. If there are already too many specialized entry points for a given callee, a generic entry point is obtained instead. This does not matter to the caller and occurs rarely in practice.

Propagating types from callers to callees allows eliminating redundant type tests in the callee, but also makes it possible to pass arguments without boxing them, thereby reducing the overhead incurred by function calls. Note that our approach does not use any dynamic dispatch to propagate type information from callers to callees. It relies on information obtained from typed shapes to give us the identity of callees (both global functions and object methods) for free. When the identity of a callee is unknown, a generic entry point is used.

In the case of the linked list example from Section 3.3, we can specialize the `sumList` function entry point based on the type tag of the `lst` parameter. As a consequence, we know whether `lst` has tag `null` or `object` upon entering the function.

With entry point versioning, we can eliminate all type tag checks, except for the check on the return type of the `sumList` call. This test seems redundant, considering that, in our example, the `sumList` function only ever returns `int32` values. The following section will explain our strategy to optimize this.

4.3 Call Continuation Specialization

Achieving full interprocedural type propagation demands passing the return type information from callees to callers. While it is fairly straightforward to establish the identity of the callee a call site will jump to in the majority of cases, establishing where a `return` statement will jump to is less straightforward. This is to say, most call sites are monomorphic and jump to a single function, and hence, a single specialized entry point. Furthermore, versioning code based on object shapes has the net effect that it will often split polymorphic call sites into monomorphic ones, which is very convenient for us.

We would like to version call continuations (the code executed when we return from a call) in accorance with the return types observed during execution. However, one `return` statement can potentially jump to several call continuations within a program. This means we cannot employ the same strategy as with entry point versioning. We cannot simply jump from one `return` statement to a specialized call continuation which assumes a known return type. Type information about return values could be propagated with a dynamic dispatch of the return address indexed with the result type. However this would incur a run time cost. We would be trading one form of dynamic overhead (that of type checks) for another (that of dynamic dispatch).

Instead, we have chosen to extend BBV with an approach that has zero run time cost (amortized overhead). Call continuations are compiled lazily when the first return to a given continuation is executed. When a function first executes a `return` statement, its return type, if known, is memorized. Call continuations are then speculatively optimized based on this memorized return type. If later returns from this function turn out to have a different return type, the optimized call continuations are invalidated (see Section 4.3.1).

Given the small example given in Figure 10 where a function `f` calls some function `g`, where `g` always returns values of type `int32`, the call continuation specialization process continuations takes the following steps:

- A call to `g` is encountered. Assuming the identity of the callee is known from typed shapes (otherwise this optimization is not performed), `f` is added to a list of callers of `g`.

```
function f() {
    // Call site
    var r = g()

    // Call continuation
    // The addition has an implicit type check
    return r + 2
}

function g() {
    return 1
}
```

Figure 10 Function call with a fixed return type.

- A stub is generated for the call continuation in `f`.
- Machine code for the call site is generated, it is made to jump directly to a specialized entry point in `g`.
- Execution resumes in `f` and jumps to `g`. Execution continues until `g` returns.
- Compilation resumes. The compiler has determined that `g` returns an `int32` value since the function `g` is annotated to indicate that it returns `int32` values.
- Execution resumes and `g` returns to the call continuation in `f`. The call continuation stub is executed.
- The call continuation in `f` is compiled. The compiler sees that `g` has been annotated as returning `int32` values. The code in `f` is optimized using this type information. No type check is performed at the addition.

The call continuation specialization process presented so far is able to optimize recursive calls in the `sumList` example and eliminate the type tag check on the return value. However, as explained in the next section, this process is speculative and does not work for every function.

4.3.1 Invalidating Call Continuations

The `makeList` function from Figure 4 is an example where the speculative call continuation specialization process fails. This is because `makeList` can return both objects and `null` values. As such, we cannot specialize callers of the function based on a single return type tag. In this situation, the speculative call continuation specialization process will try to specialize continuations, fail, and deoptimize them.

The first time that the `makeList` function returns, it will return a `null` value. This first return will then trigger the compilation of a specialized call continuation which assumes the return type of `makeList` to be `null`. When the function later returns a value with type tag `object`, this will be detected at code generation time. Callers of the `makeList` function will then have their call continuations deoptimized.

The deoptimization is done simply by writing stubs over already compiled call continuations. Should another `makeList` call return to a deoptimized call continuation, the stub will trigger the compilation of a new continuation. This time, the return type will not be specialized, because we know that `makeList` can return values with multiple type tags.

The speculative optimization and deoptimization process we employ could be seen as wasteful. We could have employed a static analysis instead. However, it can be difficult to establish the return type of a JS function simply by analyzing its code. Furthermore, the speculative approach can be more precise than a static analysis, because it is able to

take the run time behavior of code into account. The `return` statements which are never executed will not be taken into account. A static analysis does not know exactly which `return` statements are executed and which are not, but BBV does.

5 Evaluation

This section reports on an empirical evaluation of interprocedural basic block versioning. This evaluation was carried out based on an implementation of the extensions presented in this paper, namely typed shapes, entry point specialization, and call continuation specialization, within the Higgs JavaScript compiler.

A total of 26 industry benchmarks were selected from the SunSpider and V8 suites. The authors decided not to use the JSBench benchmarks [29] as they are more suited to fast interpreters (they are short running and have little computation). Benchmarks for which performance hinges on compiling regular expressions were omitted, as this is not a feature supported by the Higgs compiler.

To measure steady state execution time separately from compilation time in a manner compatible with Higgs, V8, SunSpider, TraceMonkey, and Truffle/JS, the benchmarks were modified so that they could be run in a loop. Warmup iterations are first performed so as to trigger JIT compilation and optimization of code before timing runs take place. Unless otherwise specified, 1000 warmup iterations and 100 timing iterations are used.

V8 version 3.29.66, SpiderMonkey version C40.0a1, TraceMonkey version 1.8.5+ and Truffle/JS v0.9 were used for performance comparisons. Tests were executed on a system equipped with an Intel Core i7-4771 CPU and 16GB of RAM running Ubuntu Linux 14.04. Dynamic CPU frequency scaling was disabled to ensure reliable timing measurements.

5.1 Method Identity

The extended version of Higgs tracks object shapes. Without them, the compiler would not be able to dermine which method is invoked at a call-site. With typed shapes, on average, the identity of the callee method is known for 90% of calls executed dynamically. When entry point versioning and call continuation specialization are performed, that number increases to 97.5% of calls. In practice, the identity of callees is known for most call sites. The exceptions are dominated by implementation limitations of the current version of Higgs, which currently treats captured closure variables as having unknown type.

5.2 Type Tests

Figure 11 shows the proportion of type tag tests eliminated with different variants of basic block versioning. These numbers measure actual tests executed at runtime rather than tests occuring in the program text. The first column (intra BBV) is the baseline, the number of tests that could be eliminated with plain intraprocedural basic block versioning [14]. The second column (typed shapes) shows the results obtained by adding support for typed object shapes. The third column (entry spec) adds entry point specialization, and lastly, the fourth column (entry+cont spec) adds call continuation specialization. On average, the baseline eliminates 61% of tests, typed shapes increases this to 79%. Entry point specialization improves the result to 89%. Finally, the addition of call continuation specialization allows the elimination of 94.3% of dynamic tests, and, in several cases, nearly 100%.

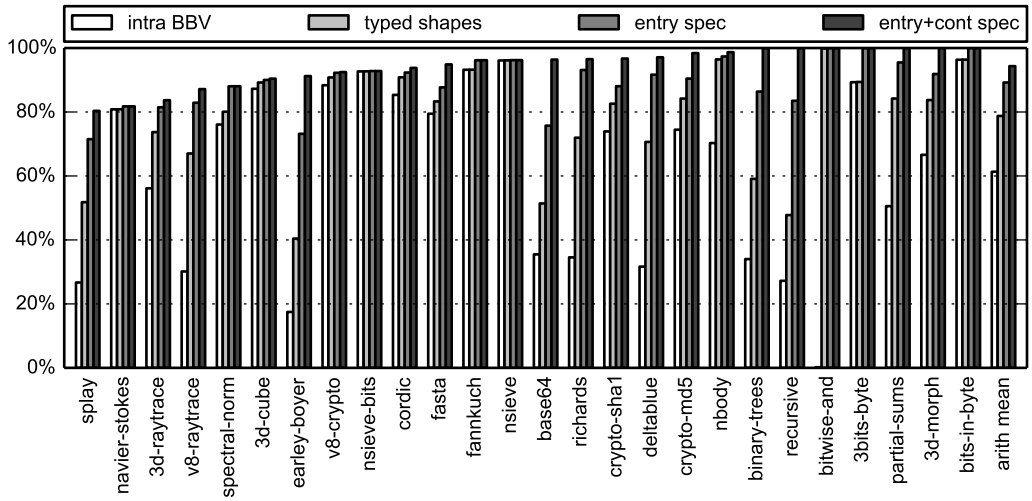


Figure 11 Proportion of type tests eliminated (higher is better).

5.3 Type Analysis

An obvious alternative to type propagation with interprocedural basic block versioning would be to perform a whole-program type analysis. As there are many different analyses in the literature with different degrees of precision, it is unclear how to evaluate the relative benefits of this paper's approach. It is possible to side-step the question by implementing an idealized static analysis. Each benchmark was run and the result of all tests was recorded. The benchmarks were then rerun with all type tests that always evaluate to the same result removed. The second run can be seen as an upper bound for the power of static analysis by itself. No static analysis can eliminate more tests than one that knows in advance the outcome of each of them. Figure 12 compares interprocedural basic block versioning and the idealized type analysis. The fact that basic block versioning outperforms type analysis should not come as a surprise. An analysis loses precision when control flow merges whereas basic block versioning creates separate versions to avoid this. The results suggest that no analysis can eliminate more than an average of 91.4% whereas Higgs can avoid executing 94.3% of tests. On more than half of the benchmarks, the proportion of eliminated tests exceeds 95%. In all benchmarks at least 80% of tests are removed.

5.4 Execution Time

The execution times of the benchmarks normalized to the unmodified version of the Higgs compiler [14] appear in Figure 13. With the exception of `navier-stokes`, `nsieve` and `nsieve-bits` (which are marginally slower), all benchmarks exhibit improvements. The largest speed up comes from the addition of typed object shapes, they improve execution time by an average of 26.8%. The addition of entry point specialization further improves performance, with a combined speedup of 36.3%. Finally, adding call continuation specialization brings the total improvement to 37.6%. The performance improvements brought by continuation specialization are relatively modest compared to those from entry point specialization. This is to be expected since entry point specialization allow us to eliminate more type tests (Section 5.2).

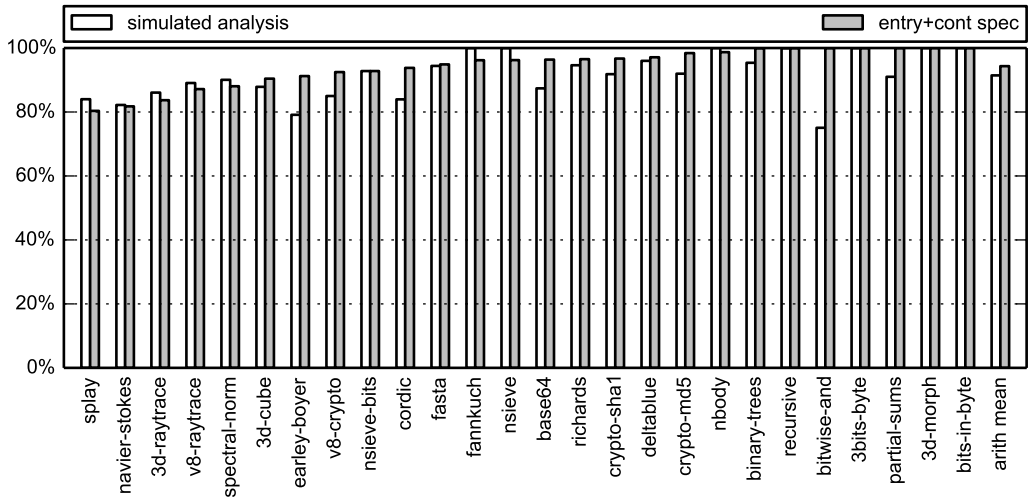


Figure 12 Proportion of type tests eliminated with BBV or a type analysis (higher is better).

5.5 Shape Tests

Our implementation of typed shapes is able to propagate known object shapes from one property access to another. There are many instances where multiple property reads on the same object occur within a given function, and shape propagation can allow eliminating further shape tests after the first property access on an object. Enabling typed shapes results in an average decrease of 27% in the number of shape tests over an unextended implementation of Higgs which uses untyped shapes and inline caches.

5.6 Call Continuation Specialization

Call continuation specialization uses a speculative strategy to propagate return type information. Call continuations for a given callee may be recompiled and deoptimized if values are returned which do not match previously encountered return types. Empirically, only 2% of functions executed cause the invalidation of call continuation code. Dynamically, the type tag of return values is successfully propagated and known to the caller 72% of the time. In over half of the benchmarks, the type tag of return values is known over 99% of the time.

5.7 Code Size and Compile Time

Adding entry point and continuation specialization to the unmodified Higgs compiler cause an increase in generated machine code size of 5.5% in the worst case and just 1.0% on average. Intuitively, one may have expected a bigger code size increase given that entry point versioning can generate multiple entry points per function. However, better optimized machine code tends to be more compact. Compile time increases by 3.7% in the worst case and just 0.01% on average.

5.8 Tracing Compilation

Tracing compilation bears important similarities to basic block versioning. One could expect tracing to do better because it can optimize long linear sequences of code. Tracing compilation

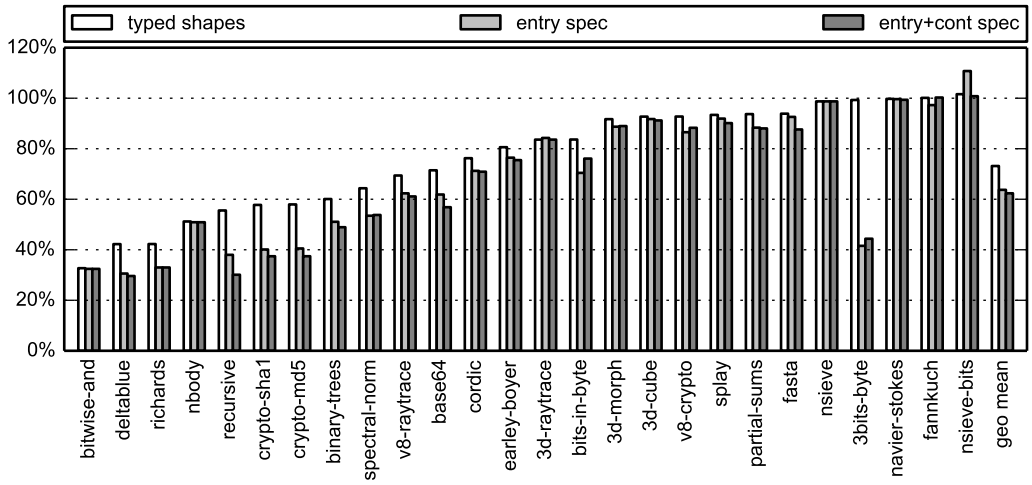


Figure 13 Execution time relative to baseline (lower is better).

was introduced to JavaScript with the TraceMonkey [17, 34] compiler. This compiler was in production within Mozilla's browser until 2011. Figure 14 compares the performance of the two compilers. On average, Higgs is 2.7x faster than TraceMonkey, and performs better on 22 out of 26 benchmarks. The benchmarks TraceMonkey achieves the best performance on tend to be ones which feature short and predictable loops.

The difference between the two is striking. It should be noted that TraceMonkey was built by a considerably larger team and implements strictly more optimizations than Higgs. For instance, it can inline while recording a trace. Even without inlining, Higgs does much better on the largest benchmarks. The two raytrace benchmarks, for instance, make significant use of object-oriented polymorphism and feature highly unpredictable conditional branches. The `earley-boyer` benchmark is the largest of all and features complex control-flow. The `splay` and `binary-trees` benchmarks apply recursive operations to tree data structures. We note that Higgs performs much better than TraceMonkey on the `recursive` microbenchmark which suggests TraceMonkey handles recursion poorly. While we caution against drawing definitive conclusions, it does appear that tracing compilation in the form implemented by TraceMonkey is mostly beneficial for computation with hot and predictable loops. Whereas Higgs is agnostic to the vagaries of control flow. It is worth mentioning that independent analysis of the behavior of real-world JavaScript programs suggests that hot and predictable loops are rare [30] and that TraceMonkey does not speed up real-world JavaScript programs such as the Google website [29].

5.9 Truffle/JS

Another interesting comparison is to look at the Truffle system from Oracle labs. Truffle/JS is an implementation of JavaScript written in Java and running on a modified Java virtual machine. Like Higgs, Truffle is a research prototype, but one being built by a larger team and with a code base about 6 times larger than Higgs'. It benefits from optimizations that are lacking in Higgs, such as method inlining and a sophisticated register allocator. For memory management it can defer to Java's highly tuned garbage collector.

Figure 15 shows the results of a performance comparison of Higgs against Truffle/JS.

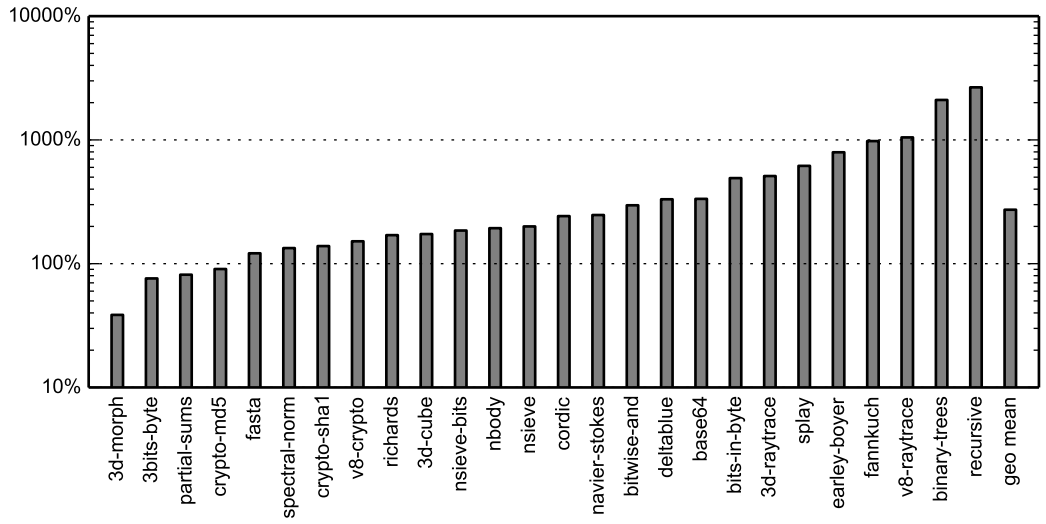


Figure 14 Speed relative to TraceMonkey (log scale, higher favors Higgs).

After both systems have gone through 1000 warmup iterations, Higgs is on average 69% as fast as Truffle/JS. The time recorded on the `3bits-byte` benchmark is zero, suggesting that Truffle used side effect analysis to optimize-away the computation.

Higgs and Truffle/JS, being research virtual machines, were not optimized for fast compilation. As a result, both systems are much slower than other engines when it comes to compilation times. We cannot directly measure the compilation time taken by Truffle/JS, but we can use the time it takes to warm up as a rough approximation.

Figure 16 shows the speed of Higgs relative to Truffle/JS when measuring the total time taken for 1000 iterations of our benchmarks, with no separate warmup iterations. On average, Higgs is 220% as fast as Truffle/JS on this comparison, indicating that the warmup and compilation time for Higgs is much shorter. This is not surprising, since Higgs begins generating type-specialized machine code as soon as program execution begins.

6 Conclusion

Basic block versioning is a compilation strategy for generating type-specialized machine code on the fly. This paper demonstrates how to extend this technique to propagate information across method call boundaries, both from callers to callees and from callees to callers, without requiring dynamic dispatch and without a separate type analysis pass.

Across 26 JavaScript benchmarks, interprocedural basic block versioning eliminates, on average, 94.3% of type tests. This is more than a static type analysis with access to perfect information could achieve. The proposed extension provides an average execution time reduction of 37.6% over an unextended basic block versioning implementation.

There is room for future work. While interprocedural basic block versioning yields encouraging results, more could be done. Two extensions to basic block versioning are planned: tracking types of closure variables and tracking array types. The Higgs compiler itself currently lacks several optimizations used by commercial virtual machines. While they are orthogonal to this paper, these optimizations may close the performance gap with commercial systems. The first optimization to add is method inlining. Inlining is likely

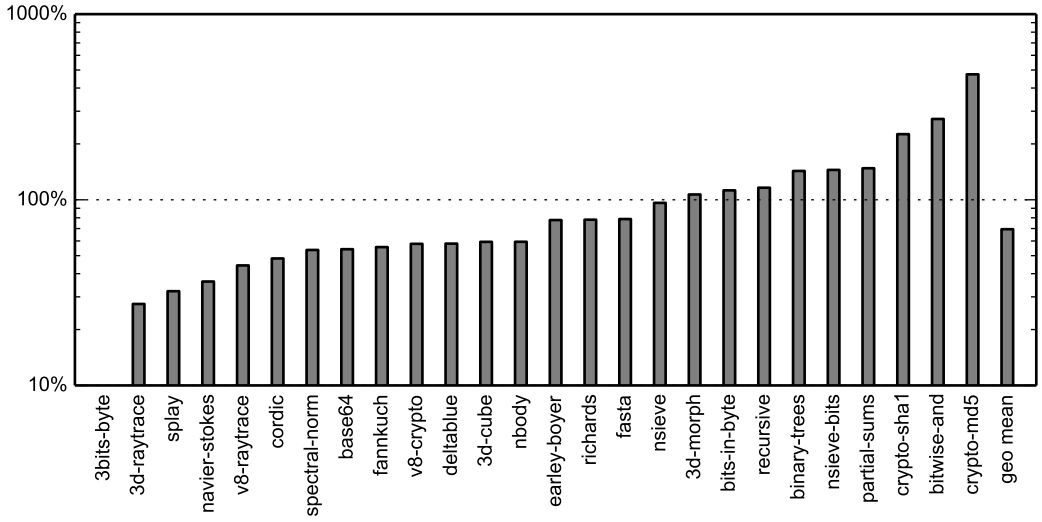


Figure 15 Speed relative to Truffle/JS (log scale, higher favors Higgs).

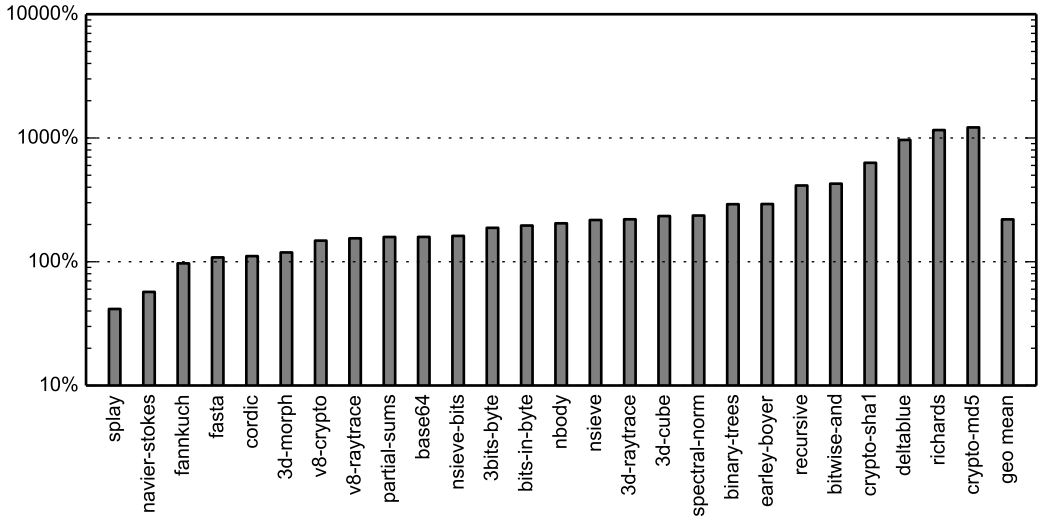


Figure 16 Speed relative to Truffle/JS, no warmup iterations (log scale, higher favors Higgs).

synergistic with basic block versioning as it provides more contextual information but it runs the risk of increasing code size as versions proliferate. Bloat can be mitigated by lazy, incremental, inlining where basic blocks are only added when needed. This would be faster than inlining entire control flow graphs without needing recompilation of the entire caller at inlining-time.

Acknowledgements. Special thanks go to Laurie Hendren, Jan Vitek, Erick Lavoie, Vincent Foley, Paul Khuong, Molly Everett, Brett Fraley and all those who have contributed to the development of Higgs.

References

1 Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. The HipHop virtual machine. In *Proceedings of the 2014 conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 777–790. ACM New York, 2014.

2 Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 2–26, 1995.

3 George Almási and David Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *Proceedings of the 2002 conference on Programming Language Design and Implementation (PLDI)*, pages 294–303. ACM New York, May 2002.

4 Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 Dynamic Languages Symposium (DLS)*, pages 53–64. ACM New York, 2007.

5 Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *Proceedings of ECOOP 2005*, pages 428–452. Springer Berlin Heidelberg, 2005.

6 V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the 2000 conference on Programming*, pages 1–12. ACM New York, 2000.

7 Michael Bebenita, Mason Chang, Gregor Wagner, Andreas Gal, Christian Wimmer, and Michael Franz. Trace-based compilation in execution environments without interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 59–68, New York, NY, USA, 2010. ACM.

8 Carl Friedrich Bolz, Antonio Cuni, Maciej FijaBkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation removal by partial evaluation in a tracing JIT. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 43–52. ACM New York, 2011.

9 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

10 Carl Friedrich Bolz, Tobias Pape, Jeremy Siek, and Sam Tobin-Hochstadt. Meta-tracing makes a fast Racket. *Workshop on Dynamic Languages and Applications*, 2014.

11 C. Chambers, D. Ungar, and E. Lee. An efficient implementation of Self a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not.*, 24(10):49–70, September 1989.

12 Craig Chambers and David Ungar. Customization: optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *Proceedings of the 1989 conference on Programming Language Design and Implementation (PLDI)*, pages 146–160. ACM New York, June 1989.

13 Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proceedings of the 1990 conference on Programming Language Design and Implementation (PLDI)*, pages 150–164. ACM New York, 1990.

14 Maxime Chevalier-Boisvert and Marc Feeley. Simple and effective type check removal through lazy basic block versioning. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 101–123. Schloss Dagstuhl, 2015. `http://arxiv.org/abs/1411.0352`.

15 Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. Optimizing MATLAB through just-in-time specialization. In *Proceedings of the 2010 international conference on Compiler Construction (CC)*, pages 46–65. Springer Berlin Heidelberg, 2010.

16 Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC)*, pages 1859–1866. ACM New York, 2009.

17 Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.*, 44(6):465–478, June 2009.

18 Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual Execution Environments (VEE)*, pages 144–153. ACM New York, 2006.

19 Michael Gorbovitski, Yanhong A. Liu, Scott D. Stoller, Tom Rothamel, and Tuncay K. Tekle. Alias analysis for optimization of dynamic languages. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, pages 27–42, New York, NY, USA, 2010. ACM.

20 David Gudeman. Representing type information in dynamically typed languages, 1993.

21 Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 239–250. ACM New York, June 2012.

22 Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag.

23 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis (SAS)*, pages 238–255. Springer Berlin Heidelberg, 2009.

24 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *Proceedings 17th International Static Analysis Symposium (SAS)*. Springer Berlin Heidelberg, September 2010.

25 Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. Type refinement for static analysis of JavaScript. In *Proceedings of the 2013 Dynamic Languages Symposium (DLS*. ACM New York, 2013.

26 Madhukar N. Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, and Ben Hardekopf. Improved type specialization for dynamic scripting languages. *SIGPLAN Not.*, 49(2):37–48, October 2013.

27 Francesco Logozzo and Herman Venter. RATA: rapid atomic type analysis by abstract interpretation; application to JavaScript optimization. In *Proceedings of the 2010 international conference on Compiler Construction (CC)*, pages 66–83. Springer Berlin Heidelberg, 2010.

28 John Plevyak and Andrew A. Chien. Type directed cloning for object-oriented programs. In *Proceedings of the Workshop for Languages and Compilers for Parallel Computing (LCPC)*, pages 566–580, 1995.

29 Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated construction of JavaScript benchmarks. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 677–694, 2011.

30 Gregor Richards, Sylvain Lesbrene, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the ACM Programming Language Design and Implementation Conference (PLDI)*, June 2010.

31 Armin Rigo. Representation-based just-in-time specialization and the Psyco prototype for Python. In *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '04, pages 15–26, New York, NY, USA, 2004. ACM.

32 Baptiste Saleil and Marc Feeley. Code versioning and extremely lazy compilation of Scheme. In *Scheme and Functional Programming Workshop*, 2014.

33 Henrique Nazare Santos, Pericles Alves, Igor Costa, and Fernando Magno Quintao Pereira. Just-in-time value specialization. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

34 Rodrigo Sol, Christophe Guillon, FernandoMagno Quintão Pereira, and Mariza A.S. Bigonha. Dynamic elimination of overflow tests in a trace compiler. In Jens Knoop, editor, *Proceedings of the 2011 international conference on Compiler Construction (CC)*, pages 2–21. Springer Berlin Heidelberg, 2011.

35 Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An object storage model for the Truffle language implementation framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 133–144, New York, NY, USA, 2014. ACM.

36 Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 2012 Dynamic Language Symposium (DLS)*, pages 73–82. ACM New York, 2012.

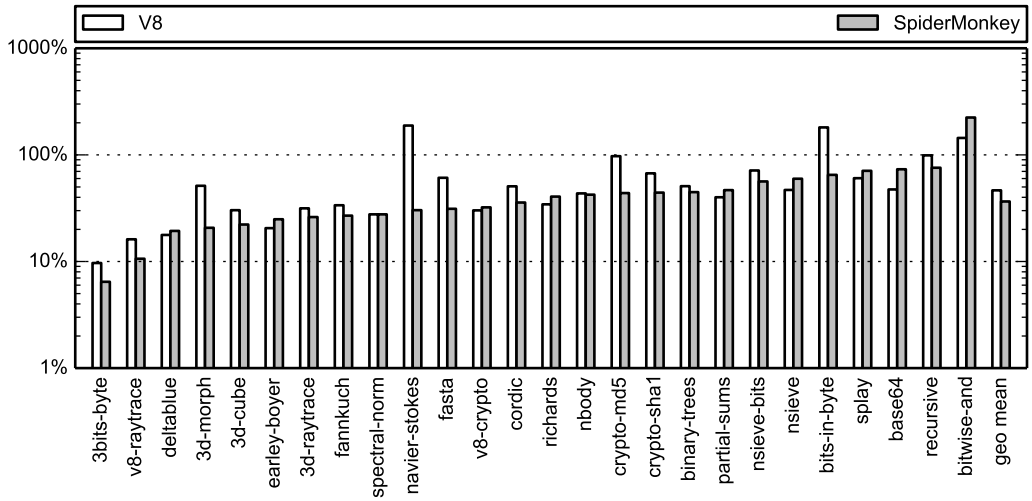


Figure 17 Speed of relative to commercial JS engines (log scale, higher favors Higgs).

A Comparison with V8 and SpiderMonkey

Figure 17 compares the speed of Higgs to optimized commercial JavaScript virtual machines. Higgs is generally slower, sometimes by an order of magnitude. There are a few benchmarks where it outperforms V8. Notably, `bits-in-byte` features many function calls, and Higgs is able to optimize this fairly well. The `bitwise-and` microbenchmark is also interesting because it is a loop performing global object property accesses. Higgs outperforms every JS engine we have tested on this benchmark, suggesting that it has faster global property accesses, thanks to typed shapes. On the other hand, Higgs is slower everywhere else. This is probably because Higgs lacks orthogonal optimizations such as: loop-invariant code motion, global value numbering, bounds check elimination, automatic SIMD vectorization, method inlining, allocation sinking, floating-point register allocation, etc. In the absence of these optimizations, BBV is most promising for use in a baseline JIT compiler.

C++ const and Immutability: An Empirical Study of Writes-Through-const*

Jon Eyolfson[1] and Patrick Lam[2]

1 **University of Waterloo**
Waterloo, ON, Canada
jeyolfso@uwaterloo.ca
2 **University of Waterloo**
Waterloo ON, Canada
patrick.lam@uwaterloo.ca

Abstract

The ability to specify immutability in a programming language is a powerful tool for developers, enabling them to better understand and more safely transform their code without fearing unintended changes to program state. The C++ programming language allows developers to specify a form of immutability using the `const` keyword. In this work, we characterize the meaning of the C++ `const` qualifier and present the ConstSanitizer tool, which dynamically verifies a stricter form of immutability than that defined in C++: it identifies `const` uses that are either not consistent with transitive immutability, that write to mutable fields, or that write to formerly-`const` objects whose `const`-ness has been cast away.

We evaluate a set of 7 C++ benchmark programs to find writes-through-`const`, establish root causes for how they fail to respect our stricter definition of immutability, and assign attributes to each write (namely: synchronized, not visible, buffer/cache, delayed initialization, and incorrect). ConstSanitizer finds 17 archetypes for writes in these programs which do not respect our version of immutability. Over half of these seem unnecessary to us. Our classification and observations of behaviour in practice contribute to the understanding of a widely-used C++ language feature.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases empirical study, dynamic analysis, immutability

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.8

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.2.1.3

1 Introduction

Immutability is an important concept that simplifies reasoning about programs and eases software maintenance. Most importantly, immutability circumscribes possible side effects, so that (in some cases) a user of a function may avoid closely examining the implementation of the function and its callees. One concrete application of immutability is: if a developer knows that a library function does not modify one of its arguments (including transitive arguments), then they know that it is safe to call that library function with that argument from multiple threads, as the function only requires read access to its argument.

* This work was supported in part by Canada's Natural Science and Engineering Research Council as well as a Google Faculty Research Award.

C++ [6] is a popular language that allows programers to specify immutability using `const`[1]. C++ experts such as Meyers recommend judicious use of "`const`-correctness" [7] in C++ codebases. It is generally clear when developers could use `const`, but not (in non-obvious cases) when they should use `const`.

We distinguish between two uses of C++'s `const` qualifier: `const`-qualified global/stack objects, whose data may never change (i.e. immutable objects), and `const`-qualified references/pointers to objects (i.e. read-only references [9]). (For now, assume that there are no casts that remove `const` qualifiers and no `mutable` storage class specifiers; these language features violate immutability.) An *immutable object*'s fields may never change, and mutable references to that object may never be created. A *read-only reference*, on the other hand, only guarantees immutability for accesses through qualified references. An object with a read-only reference to it may still be mutated through other, mutable, references. C++ enforces a shallow immutability guarantee (also known as bitwise `const`) for writes through the read-only reference: while it is illegal to reassign the fields of such an object, any referents of fields may change. We expand on this in Section 3.

C++'s type system admits several workarounds to `const`'s supposed immutability guarantees. Much research defines type qualifiers similar to C++ `const`, but with stronger guarantees. These type systems do not have any holes in the type system, such as unsafe casts. Furthermore, they not only ensure that the field values do not change, but also ensure that objects referenced through fields also do not change (i.e. deep, or transitive, immutability; again, see Section 3 for more details).

On the other hand, our industrial contacts have indicated that, in their codebases, `const` has been used to deter new developers from modifying certain variables. Such variables may be modified by an experienced developer ready to assume the consequences [4]. In such cases, `const` serves an advisory role, but does not provide any guarantees.

Our goal is to explore the space of possible meanings for immutability declarations in C++ and to examine what guarantees developers appear to be expecting in practice. We developed a tool, ConstSanitizer, that instruments programs to identify source locations that modify `const`-qualified objects, using more restrictive semantics than guaranteed by C++. ConstSanitizer monitors writes-through-`const`, i.e. writes performed on `const`-qualified objects or references, either transitively (which is allowed in C++), or through C++'s `const` escape hatches. To better understand `const` usage in practice, we ran ConstSanitizer on a benchmark suite and manually classified all writes-through-`const`.

Specifically, the goal of this work is to answer the following research questions:

(RQ1) Do developers perform shallow and transitive writes-through-`const`?
(Answer: Yes to both.)

(RQ2) How do developers write-through-`const`?
(Answer: By directly writing to fields of `const`-qualified objects and through transitive writes; both are about equally common.)

(RQ3) Why do developers write to fields of `const`-qualified objects?
(Answer: Buffers and delayed initialization were important reasons, but over half the time, we couldn't find any clear reason motivating developers' decisions to write through `const`.)

Section 8 presents our detailed answers to these questions.

[1] Some of our discussion also applies to C, but we focus on C++ in this paper.

Listing 1 Method `evil()` violates the spirit of `const` by causing a write to an externally-visible field of `const` object "`a`". Circled numbers used for subsequent explanations.

```
class A { public: int id; };
size_t std::hash(const A& a) { return std::hash(a.id); }
std::unordered_map<A, std::string> m;
(1) const A a;

m.insert(a, "Value");
(2) evil(a);
m.find(a);

                                        (3)
void evil(const A& a) { writeId(const_cast<A*>(&a)); }
(4) void writeId(A *pa) { pa->id = 5; }
```

Our contributions include:

- the design and implementation of a novel dynamic analysis for C++ that detects writes-through-`const`-qualified variables (both shallow and transitive);
- an empirical study of `const` usage (including writes-through-`const`-qualifiers) on a suite of 7 C++ benchmarks; and,
- based on the empirical study, a novel classification of writes-through-`const`-qualifiers in the wild according to a root cause and a set of attributes.

2 Motivating Example

We continue with an example of a `const` usage that must be accepted by C++ compilers [6] but leads to undefined behaviour when used with the C++11 standard library specification.

Listing 1 contains function `writeId()`, which modifies field `id` of its parameter. Function `evil()` takes a `const`-qualified parameter, casts away the `const` qualifier, and calls `writeId()`. Both these functions must be accepted by C++ compilers.

Function `writeId()` does not perform anything unexpected. The write to `id` is legitimate with respect to `const`: `writeId()` has a non-`const` reference to data and is therefore entitled to write to it. (`writeId()` conflicts with a different part of the library specification—one oughtn't write to fields used as hashes—but that conflict is beyond the scope of this work.)

Function `evil()` accepts a `const` parameter "`a`"; the `const` qualifier intuitively suggests that the state of "`a`" should not change across a call to `evil()`. (Section 3 explains the C++ semantics of `const` in more detail; Section 4 explains the writes that ConstSanitizer monitors.) `evil()` then casts away the `const` qualifier and calls `writeId()`, which writes to the `id` field, thus changing the state of "`a`".

Listing 1 also contains client code. This client code provides a `hash()` function for when objects of type "`A`" are used with the standard library. It continues with a declaration of an `std::unordered_map` `m`, which gets a `const`-qualified object "`a`" added to it. (In C++, objects like "`a`" and `m` are constructed upon declaration.) Between the `insert()` call and the `find()` call, the program calls `evil()`, which changes the `id` field, thus causing the `hash()` of "`a`" to change. As a result, the `find()` call may unexpectedly return `m.end()`, indicating that it did not find "`a`" in the expected bucket; that result should be a surprise to the client.

In our example, the client code is doing nothing wrong, yet it may get an unexpected result from the map. The client should be entitled to believe that its `const`-qualified "`a`" object does not change between the two map calls and that the object is still in the map.

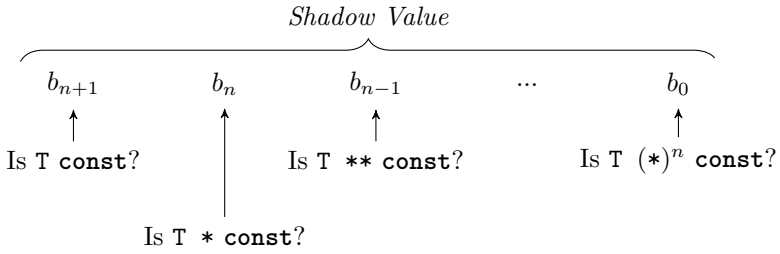


Figure 1 Our shadow values encode the `const`-ness of each level of an n level pointer.

Analysis of example. We informally describe how ConstSanitizer works on our example. Our LLVM-based tool actually operates at the `llvm` bitcode level (assisted by metadata from `clang`), but for clarity we describe shadow values and the effects of statements on them using C++ code. Section 4 describes our analysis as-implemented on top of LLVM.

ConstSanitizer associates a shadow value with each variable. This shadow value describes whether or not the variable, and each of its dereferences, may be written to; Figure 1 shows how shadow values encode `const`-ness. We present the semantics of our shadow encoding more thoroughly in Table 1. ConstSanitizer propagates shadow values through the program's execution. At each write, ConstSanitizer verifies that the shadow value permits a write, and indicates a write-through-`const`-qualifier if not.

We next show how ConstSanitizer works; circled numbers refer back to Listing 1.

(1) Program allocates `const`-qualified new object "`a`". Using debug information, ConstSanitizer finds the `const` qualifier in "`a`"'s type, and associates shadow value $(1)_2$ with "`a`", indicating that "`a`" is `const`.

(2) ConstSanitizer then propagates shadow value $(1)_2$ to the function call `evil()`. Inside function `evil()`, variable "`a`" is a reference, so we shift the shadow value to the left and obtain $(10)_2$ inside the function call boundary.

(3) Program casts from type `const A*` to type `A*`. Because "`a`" is a reference inside `evil()`, the address-of operation has no effect on the shadow value. (On a non-reference, taking the address would also result in a logical shift left of the shadow value.)

The cast of the `const` qualifier is invisible to LLVM, so ConstSanitizer propagates shadow value $(10)_2$ across the cast. (Had "`a`" originally not been `const`, a pointer to it would have shadow value $(00)_2$ and it would have shadow value $(0)_2$.)

Finally, ConstSanitizer passes shadow value $(10)_2$ for parameter "`pa`" of `writeId()`.

(4) Inside function `writeId()`, the instrumented write to field "`id`" observes that the shadow value of the address of the containing object is $(10)_2$. Because the left-hand side uses the `->` operator, ConstSanitizer shifts the shadow value back right once, giving shadow value $(1)_2$ for the containing object. Because the containing object is `const`, we also apply a shadow value of $(1)_2$ to all writes to fields of that object.

When executing the write, ConstSanitizer checks the right-most bit of the shadow value for the destination. Since this value is $(1)_2$, the program is writing a value through a `const` reference. ConstSanitizer therefore signals a write-through-`const`.

Section 5 contains our classification of writes-through-`const`. We classify this write as root cause "C", a write after casting away a `const`, and assign attribute "I", for incorrect.

Listing 2 The `const` qualifier may apply to C++ member functions.

```
class Pointish {
private:
  int x;
  int * y;
public:
  int getX() const { return x; }
  void setX(int val) { x = val; }
  void setY(int val) { *y = val; }
};
```

not OK if `setX()` were `const`

transitive write
OK if `setY()` were `const`

3 Meaning of `const` in C++

As discussed earlier, the C++ **`const`** keyword allows programmers to declare, in some sense, that a value should not change. In this section, we explain the specific guarantees that C++ provides, and we present the deep immutability variant of these guarantees that we verify.

Meaning of `const` on C++ primitive and pointer types. The meaning of **`const`** in C++ is an extension of its meaning in C. We start by describing the common meaning of **`const`** across C and C++, applying to primitive and pointer types. In these languages, the **`const`**-ness of a memory location depends on the qualifiers of the variable through which the location is accessed; our motivating example illustrated a change in **`const`**-ness through a cast, in function `evil()`, that is allowed by both C and C++.

Developers may **`const`**-qualify primitive types such as `int`, resulting in immutable object types like `const int`. When variable v has primitive **`const`**-qualified type, the C++ type system prevents developers from assigning to v after its definition; i.e. it prevents writes to v. In this case, **`const`** behaves like `final` in Java, which also prevents re-assignment.

For pointer types, such as `int *`, developers may **`const`**-qualify both the pointee and pointer type. The **`const`** qualifier applies to the type directly to the left of it; if there is nothing to the left, then **`const`** applies to the right. If a variable has type `int *const`, developers may not change the address value of the variable (where it points to), but they may dereference the location and change the value it points to. A different type is `int const *` (also known as `const int *`), which allows the address value to change, but not the value pointed-to by the variable. This type represents a read-only reference. The **`const`** qualifier may also apply to both the pointer and pointee types (`int const * const`), which prevents writes to both the address value and the value pointed-to. If all pointers to a value are read-only references, then the value pointed-to is immutable. C++ references can be thought of as **`const`**-qualified pointers; a developer may not write to the reference address value. However, in contrast with pointers, developers cannot cast away reference address value **`const`**-ness and re-assign the address value; this property is enforced by the language.

Meaning of `const` on C++ object types. We continue by exploring the meaning of **`const`** in C++-specific contexts. When a C++ object type is **`const`**-qualified, the developer may only call member functions declared with a **`const`** qualifier.

`const`-qualifying a member function has two effects. First, **`const`**-qualifying a member function allows it to be called on a **`const`**-qualified receiver object. Furthermore, inside the function, the type qualifiers of the receiver object's fields are treated as **`const`**.

Conceptually, each C++ class provides two interfaces: the **`const`**-qualified interface and the non-**`const`** qualified interface. A **`const`**-qualified reference is meant to be a read-only reference, although C++ enforces no guarantees. One of our goals is to evaluate whether read-

only guarantees hold in practice. When an object has non-`const` type, then the developer may call all methods on that object[2]. On the other hand, when an object has `const` type, then the developer may only call methods on that object that are `const`-qualified.

Consider class `Pointish`, defined in Listing 2. As written, the developer could call all methods on a non-`const`-qualified object of type `Pointish`. On the other hand, the developer may only call the `getX()` method on a `const`-qualified `Pointish` object.

The second effect of `const`-qualifying a function changes the type qualifiers of fields inside the function. In our example, field `x` becomes `int const` within `const`-qualified member functions. The compiler successfully compiles `getX()`, since there are no writes to `x` or `y`. But, if `setX()` was `const`-qualified, the compiler would refuse to compile the code, since the type of `x` would be treated as `const int` and `setX()` contains a write to that variable.

In C++, without using `const` escape hatches, developers may re-assign fields in non-`const` qualified methods and may not re-assign fields in `const`-qualified methods. In all methods, developers are permitted to mutate state outside of re-assignment (through references or pointers). This type of immutability is referred to as *shallow immutability*.

A C++ `const`-qualified stack/global object would be considered a shallow *immutable object*. That is, without escape hatches, developers cannot create non-`const` references (including through pointers) to such a `const`-qualified object. However, as we discuss next, developers may indeed remove the `const` qualifier on references to the `const`-qualified object. Therefore, C++ does not strongly enforce the concept of an immutable object.

Working around `const` restrictions. Practical type systems appear to require escape hatches. C++'s escape hatches for `const` include casting (the sole escape hatch in C) and `mutable`. Also, C++ `const` does not specify deep immutability. ConstSanitizer dynamically observes executions to monitor uses of escape hatches and deep immutability.

Most type systems permit casting between types. C-style casts (`(const A)a`) and C++ `const_cast`s can add or remove `const` qualifiers. `const` manipulation may also occur through unions and `reinterpret_cast`s. ConstSanitizer ignores casts, instead using the declared type of a variable or function argument. When there is a mismatch between variable and function argument types, we persist all `const` information.

For `const` member functions, "`mutable`" instructs the compiler to not add the implicit `const` type qualifier otherwise imposed on fields inside those functions. In Listing 2, if `x` were instead declared as `mutable int x`, then `setX()` could be `const`-qualified and still compile. ConstSanitizer would then report the write to field `x` whenever `setX()` was called on a `const` receiver object, as we consider the write to be a breach of that object's immutability.

ConstSanitizer goes beyond C++'s guarantees with respect to transitivity. Consider again Listing 2. Field `y` has type `int *`; within a `const`-qualified member function, its type is `int *const`. C++ therefore prevents writes to `y` inside a `const`-qualified member function. However, it does not prevent writes to `*y` without further explicit markup (i.e. `int const *const`). We call writes to locations like `*y` *transitive writes*.

C++ only guarantees shallow immutability—that is, field values directly stored in a `const`-qualified class do not change. If a field has pointer type, C++ ensures that the pointer value does not change, but does not guarantee anything about the value pointed to. ConstSanitizer verifies *deep immutability* through transitive writes, and enables us to answer the empirical question of whether extant programs preserve deep immutability or not.

[2] There is a small exception: on a non-`const` object, the developer cannot call `const`-qualified methods that are hidden due to overloading by a non-`const`-qualified method of the same signature.

Listing 3 C++ source code showing a false negative due to expression handling.

```
const int * x = new int(0);
int * y = const_cast<int *>(x);
*y = 1;
```
not reported

4 Technique

Our ConstSanitizer tool generates instrumented code which, when executed, prints out notifications about writes-through-**const**-qualifiers[3]. ConstSanitizer builds upon LLVM [16] and was inspired by existing sanitizers including AddressSanitizer [13] and MemorySanitizer [14].

We implemented ConstSanitizer by extending the `clang` frontend and adding instrumentation passes on `llvm` bitcode. The instrumented code calls hooks in our modified version of `llvm`'s `compiler-rt` runtime library. Figure 2 depicts our processes for compiling instrumented code. Plain text indicates inputs and outputs; outlined boxes indicate existing software components; and light gray boxes indicate our modifications.

We first describe our modifications to the `clang` frontend. When the developer enables ConstSanitizer (using a command-line flag), our frontend adds metadata about initialization expression extents to the bitcode. This metadata notifies the `llvm`-level instrumentation about source-level constructs that would otherwise be lost in translation to bitcode.

Specifically, we modified `clang`'s bitcode generator at variable declaration statements. At statements of the form `type var = expr` we mark the instructions making up `expr` so that our `llvm`-level instrumentation can ignore them. The rationale for ignoring those writes is that the primary user-visible write from a `clang` declaration statement is to `var`, on the left-hand side. We empirically observed that other writes within `expr` are almost always initialization writes to `var`, which ought not to be reported even if `var` is **`const`**. Although a programmer may include explicit (side-effecting) writes within `expr`, we ignore such writes to eliminate the false positives that otherwise occur due to initialization writes.

We use **`const`**-ness information as provided by declarations, rather than implementing a taint-based approach. Listing 3 shows a false negative caused by our approach. The debugging information for `y` gives shadow value $(00)_2$. Hence, on the write through `y`, we do not report a write-through-**`const`**, because we do not propagate **`const`**-ness information from variable initializers. One might expect this write to trigger a report since `y` aliases the read-only reference `x`. (We report a write if the cast is part of a function argument.)

Most of our instrumentation lives in a custom `llvm` pass that generates code to track **`const`**-ness of the program's values. The instrumentation manipulates shadow values to track **`const`** qualifiers at every instruction that generates a pointer value. The **`const`** information relies on type tables from DWARF 4 debugging information. In `llvm`, this includes all variables in functions—all local variables are allocated on the stack and are pointers.

Our dynamic analysis returns (as one might expect) no false positives, since it observes program executions. However, it depends on the accuracy of the debugging information and metadata, which it uses to identify which variables are **`const`**-qualified in the source and to identify initialization expression extents. We ran into one false positive in our results which we believe is the result of the metadata being invalidated between LLVM passes.

[3] We previously implemented a static analysis which was an unpublishable dead end. Most of its reported violations required calling context to make sense of; context-sensitive interprocedural analysis would thus be required to get meaningful results. Static counts of mutables and const casts were too overwhelming. Furthermore, imprecision due to pointers made its results unusable.

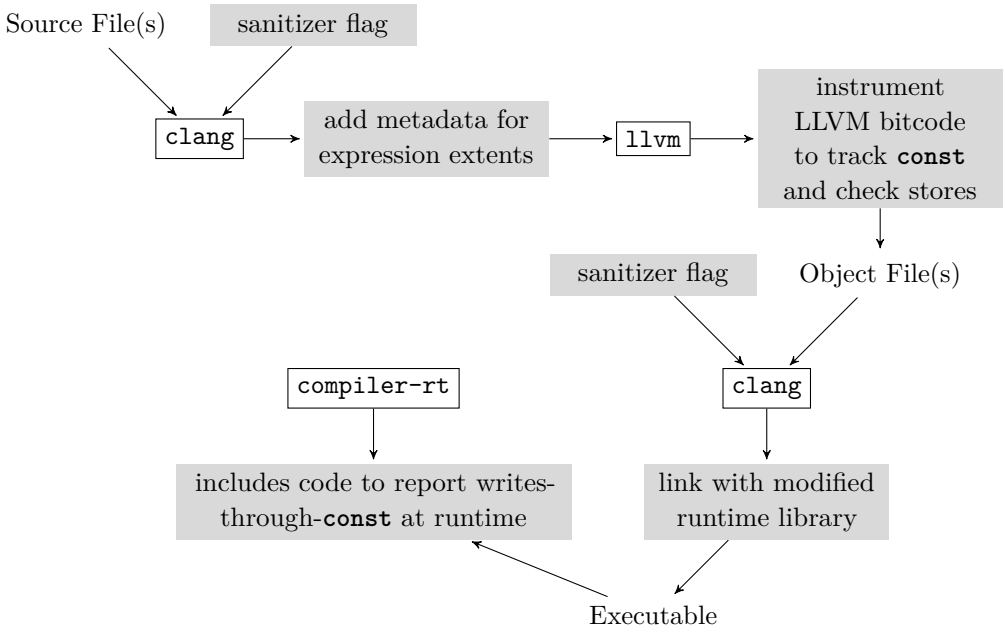


Figure 2 ConstSanitizer generates instrumented LLVM bitcode which reports writes through `const` qualifiers at runtime.

Throughout the remainder of the section, we point out a couple of cases where our analysis must approximate intended `const`-ness because actual `const`-ness information does not exist.

Structure of shadow values. A shadow value consists of n bits tracking `const`-ness (where n is the word length of the processor architecture). Each bit represents whether a pointer or pointee has a `const` qualifier or not. The rightmost bit represents the `const` qualifier of the value itself. Bits to the left (if the value is a pointer) represent what the pointer transitively points to. Our encoding supports pointers up to $n-1$ levels deep on n-bit processors (64 for our experiments). Figure 1 depicted our encoding of shadow values, while Table 1 shows how shadow values represent sample `const`-ness settings and corresponding writes allowed.

Shadow value computation. We next describe how we create and propagate shadow values. Our ConstSanitizer instrumentation dynamically propagates shadow values representing `const` qualifiers through a program's instructions. Our goal is to monitor 1) writes to `mutable` fields; 2) locations where `const` has been cast away; and 3) transitive writes, to pointees of fields, through `const` references. Table 2 summarizes the analysis rules.

`llvm` bitcode uses `alloca` instructions to introduce new pointer values. Our `llvm` pass instruments each `alloca` instruction with the appropriate shadow value, as extracted from the type information in the source code, using standard `clang` debug information.

Ultimately, our instrumentation verifies the behaviour of `store` instructions. Recall that we exclude `store` instructions that come from the right-hand side of a declaration statement. For all other `store`s, we check whether the operand—the location being written-to—represents a `const`-qualified type. The rightmost bit of the shadow value provides this information. If that bit is 1, an execution of this `store` instruction is a write to a `const`-qualified location. We insert a call to our runtime library to check the value of the bit and to report a write-through-`const` if the bit is 1. (We later discuss a special case for store instructions where the value being stored is a function argument.)

Table 1 Shadow values encode available `const`-ness restrictions on variables.

Declaration	Shadow value	Example statement	Allowed
`int x`	$(0)_2$	`x = 5`	✓
`const int x`	$(1)_2$	`x = 5`	×
`int * x`	$(00)_2$	`x = y`	✓
		`*x = 55`	✓
`int *const x`	$(01)_2$	`x = y`	×
		`*x = 55`	✓
`const int * x`	$(10)_2$	`x = y`	✓
(or `int const * x`)		`*x = 55`	×
`const int *const`	$(11)_2$	`x = y`	×
(or `int const *const`)		`*x = 55`	×

Table 2 Dynamic analysis rules showing computation of shadow value for result `%1`.

Instruction	New shadow value
`%1 = alloca ...`	from `const` qualifiers in debugging information, consistent with Figure 1.
`%1 = getelementptr %2`	by logically shifting left `%2`'s shadow value once for each dereference this instruction represents. **if field access**: check `const` qualifier of base object; for (immutable) `const` base objects, new shadow value is all ones, otherwise all zeros.
`%1 = call(%2)`	loaded from return shadow value in Thread-Local Storage (TLS). **for pointer arguments `%2`:** also write shadow values to appropriate TLS slots for the function call; if the call and argument are marked as ignored, write all zeros for the shadow value for the argument.
`%1 = phi/select ...`	carry out same operation on shadow value operands.
`%1 = bitcast %2`	from the shadow value for `%2`, if compatible; otherwise all zeros.
`%1 = load %2`	logical shift right of `%2`'s shadow value.
`store %2, %1`	check rightmost bit of shadow value for `%1`, report write-through-`const` if set. (Only applies if the instruction not ignored as an initializer.) **if `%2` is function argument:** load shadow value for `%2` from TLS, left shifted once. New shadow value is bitwise OR of shifted value with previously computed shadow value for `%1`. **if `%2` is "this" function argument:** same steps as above, except skip the bitwise OR step. **if `%2` is "this" function argument for destructor:** shadow value for `%1` is $(00)_2$.

`%1 = extractelement`
`%1 = extractvalue`
`%1 = inttoptr`
`%1 = landingpad`
	all zeros.

Conversely, `load` instructions return a pointer that represents a single pointer dereference. To compute the returned shadow value, we right shift the operand's shadow value.

`llvm`'s `getelementptr` (GEP) instruction accesses arrays and fields of objects. This instruction preserves type safety through dereferences in the compilation process and is a safe alternative to directly generating pointer arithmetic code. Our instrumentation performs a logical shift right by one bit for every pointer dereference implied in the GEP instruction. Our treatment of GEP implicitly handles transitive immutability as follows: when a GEP accesses an object field, and the containing object is `const`-qualified, we generate a shadow value as if the field had a `const` qualifer on every type for the contained field. This treatment implies checks for transitive immutability; generating a non-`const` shadow value here would generate the same bitwise immutability checks for `const` as specified by the C++ standard.

Our instrumentation propagates `const`-ness information (in shadow values) alongside references to that location. In C++, access restrictions to a location depend on whether the program is accessing that location through a `const` reference or not. Therefore, in the presence of casts and pointer arithmetic, there is no ground truth about the `const`-ness of the resulting references and we must make a reasonable under-approximation as to `const`-ness.

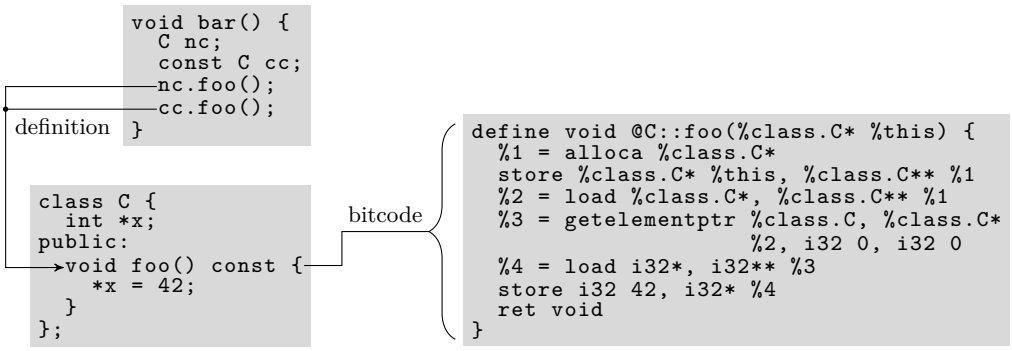
We next discuss casting-related `llvm` instructions. The `bitcast` instruction converts a value into a specified type. If a program converts a pointer between equally-indirected pointer types, then we copy the old shadow value to the result. (C++ `const`-casts do not appear at LLVM bitcode level, nor do the component of a C-style cast that manipulates `const`-ness. ConstSanitizer preserves declared `const`-ness for such variables.) Otherwise, we choose to assume that the instruction's result has no `const` qualifiers. We make this assumption in all cases for the `inttoptr` instruction, which represents pointer arithmetic not handled by the GEP instruction, as well as for `extractvalue` and `extractelement`.

Our instrumentation stores shadow values for function calls' arguments and return values using thread local storage (TLS). In the straightforward case, we store shadow values for pointer arguments in TLS slots reserved for each argument. However, we ignore pointer arguments' `const` qualifiers if the call and the argument are both part of a variable declaration. As for (pointer) return values: if a function was instrumented by our tool, then we read the shadow value from the appropriate TLS slot. We also store a mutable shadow value in the return value TLS slot, in case the function had not been instrumented by our tool. Our instrumentation either reads the approximation or, if applicable, the actual return value generated by the called function. In the presence of callbacks from uninstrumented code back to instrumented code, our instrumentation may use stale shadow values and report extraneous results based on these stale shadow values.

We have a special case for store instructions where the value stored is a function argument, as mentioned above. Consider a store instruction "`store value, location`". We compute the shadow value for "`location`" as follows. First, we get the shadow value for "`value`" from the TLS. Then, we adjust this shadow value to be compatible with the type of "`location`": our encoding requires one logical shift to match the type of "`value`" to that of "`location`". We bitwise OR the shadow value for "`location`" previously computed (from an `alloca` instruction) with the shifted value to get the new shadow value for "`location`". This preserves `const` qualifiers of the original argument and of the local variables in the function.

There are two further special sub-cases for store instructions and function arguments for (i) method and (ii) destructor calls. Listing 4 illustrates sub-case (i). Here, `foo()` is declared `const`. The compiler will hence treat `this` as `const` within `foo()`. However, for our dynamic analysis, we want to detect writes based on the `const`-ness of `this` from the caller; in method `bar()` in Listing 4, receiver object `nc` for `foo` is not `const`, so we do not want to report the call's (transitive) store to `x`. `foo`'s method arguments appear as "`value`"

Listing 4 C++ source code showing calls to method `foo()` (with its definition and associated LLVM bitcode) from `const` context `cc` and non-`const` context `nc`.



operands of `store` instructions while the "`location`" is an `alloca` within the function. We set the shadow value of the associated `alloca` instruction to the value of the argument after applying a logical shift left by one (since it's a pointer). This treatment properly ignores `const` qualifiers added due to callee method signatures.

For sub-case (ii), destructors, we do not want to report any writes through `this` as the object no longer exists after the call (so that writes to the object aren't visible in any case). We handle this case by simply assuming that the `this` argument is mutable. For all other arguments, we do a bitwise OR between the `alloca` shadow value and the argument shadow value logically shifted left by one, which maintains all `const` qualifiers.

Shadow value computation example. Listing 4 presents the C++ source code for `C::foo`, a `const` qualified method, and the associated LLVM bitcode. Consider the `bar` function, which calls `foo` twice, first with mutable (i.e. non-`const`) receiver object `nc` and then with `const` receiver object `cc`. Within `bar`, the shadow value of `nc` is $(0)_2$ and the shadow value of `cc` is $(1)_2$. Our instrumentation assigns shadow values for each LLVM instruction with a pointer result. We instrument `C::foo` as follows:

- The first instruction, `alloca`, stores its result in `%1`. Since it is an `alloca` instruction, we obtain its shadow value from `clang` debugging information. The associated shadow value is $(10)_2$: in this `const`-qualified method, the type of `this` is that of a `const` pointer to the containing class, `const C *`.
- At the `store` instruction, without special handling, we would load the shadow value of argument `%this` from the TLS; logically shift left the shadow value by one to account for the fact that we are performing a `store` to memory allocated for that argument; and bitwise OR the resulting shadow value with the original shadow value for `%1`. In our example, whether the receiver object is `cc` or `nc`, the shadow value for `%1` is $(10)_2$.
- Next, we obtain the shadow value for the result of the `load`, `%2`. As `%2` returns a pointer, we shift `%1`'s (the operand's) shadow value right by one, giving a shadow value of $(1)_2$.
- Next, the `getelementptr` instruction results in a pointer to the class's `x` field. Our instrumentation of `getelementptr` could produce two different shadow values, depending on the instruction's operand. In this case, `%2` is a `const` object, and the resulting shadow value for a fully `const`-qualified `x` field is $(11)_2$.
- Next, we obtain the shadow value for the `load` result `%4` using the same technique as for `%2`. The resulting shadow value is $(1)_2$.

Table 3 Root causes of writes through `const` and our symbols for these causes.

Root Cause	Symbol
Write to mutable field	M
Transitive write	T
Write after casting a `const` qualifier away	C

- Finally, we insert a check at the `store` instruction. In this case, the least significant bit of the shadow value associated with location (`%4`) is 1. Therefore we would dynamically report a write-through-`const` at the write to field `x` of the `const` method.

For methods, this instrumentation is not enough. We only want to report a write-through-`const` for the call with `const` receiver object even though both objects call the same static method. Before the call to `foo`, our instrumentation stores the shadow value of the receiver object in its TLS slot. Our instrumentation of `foo` looks for `store` instructions that use the receiver object and recomputes the shadow value of the location. Here, we load the shadow value from its TLS slot and shift left to match the type of the expected shadow value of `%1`. For `nc`'s call, this shadow value is $(00)_2$. Since `foo` is a method, we ignore the original shadow value of `%1` ($(10)_2$) and overwrite it with new shadow value $(00)_2$. Following the remaining steps in `foo` as above, the shadow value of `%4` is now $(0)_2$ and we do not report a write-through-`const`. In the `cc` case, we would follow the same steps, but instead report a write-through-`const`, because the shadow value would be $(10)_2$.

5 Classification of writes-through-`const`

One of our contributions is a careful analysis of the `const` usages detected by our ConstSanitizer dynamic analysis tool. We propose a classification for writes-through-`const`-qualifiers along 2 axes. We manually assigned each write 1) a single cause, from a set of common root causes; and 2) a set of additional attributes. This classification distills our empirical observations about `const` use in practice.

Table 3 lists all of the root causes for writes-through-`const`, along with a one-letter abbreviation that we will use in Section 6's tables. ConstSanitizer detects such writes and reports them to the user. The causes are:

- mutable field (M): the program writes to a `mutable`-labelled field of a `const` object.

```
class Mutable {
  mutable int x;
public:
  void mutator() const { x = 42; }
};
```

  `mutable` permits method `mutator()` to write to field `x` even though it is a `const` method, which would ordinarily prevent (at compile-time) writes to fields of the `this` object.
- transitive write (T): the program writes through a field of a `const` object.

```
class TW {
  int *x;
public:
  void transitiveWrite() const { *x = 42; }
};
```

  `const`-qualified method `transitiveWrite()` writes to field `x` of the `this` object. While the `const` qualifier prevents mutation of the `x` field, it does not prevent transitive writes of the memory pointed to by `x`.

Table 4 Observed common attributes of writes through `const` and corresponding symbols.

Attribute	Symbol
Write is synchronized	S
Write is not visible	N
Write is to a buffer/cache	B
Write is delayed initialization	D
Write is incorrect	I

- casting away const (C): the program writes through a pointer which has previously been **`const`** but whose **`const`**-ness has been cast away using a `const_cast` or C-style cast.
```
void writeToArg(int *y) { *y = 17; }
const int *x = ...;
writeToArg(const_cast<int *>(x));
```
The write in `writeToArg()` mutates the value pointed-to by `x` while `x` is **`const`**-qualified. ConstSanitizer reports writes-through-**`const`**-qualifiers whose **`const`**-ness has been cast away, using the **`const`**-ness of the most recent declared type for the value.

Table 4 summarizes our attributes for writes-through-**`const`**-qualifiers. We assigned attributes to writes based on our understanding of the code. Writes may have multiple attributes; for instance, a write in our Protobuf benchmark is B & N & S. The attributes are:

- synchronized (S): indicates that the write is always protected by a lock. This attribute is often required under the C++11 standard: all types that are shared between threads and that may be used with the standard library must be either bitwise const, which is clearly not the case when we witness a write, or else protected against concurrent accesses [15]. The following example, from Protobuf, is synchronized using Google mutex primitives.
```
GOOGLE_SAFE_CONCURRENT_WRITES_BEGIN();
_cached_size_ = total_size;
GOOGLE_SAFE_CONCURRENT_WRITES_END();
```
- not visible (N): indicates that the result of the write is never externally visible (e.g. private and with no accessor methods; may be accessed in the same translation unit). Often occurs in the context of testing-related counters.
```
mutable int countFooCalls;
void foo() const { ++countFooCalls; }
```
- to a buffer/cache (B): indicates that the write is of a derived value which can be computed from other currently-available state. Such writes are often optimizations.
```
_cached_size_ = total_size;
```
- delayed initialization (D): indicates that the write initializes state not initialized in the constructor or its transitive callees. Writes with this attribute could have also occurred in the constructor, but the written value was not yet available. Failure to call a delayed initialization method would lead to undesired behaviours (or lack of desired behaviours).
```
bool Generator::Generate(const FileDescriptor* file, ...) const {
  this->file_ = file;
}
```
- incorrect (I): indicates that the write appeared to violate the **`const`**-ness of the object.

Note that S/N/B/D writes are not necessarily errors and do not necessarily violate immutability properties. We thus chose the word "attribute" to suggest that S/N/B/D

Table 5 We ran our experiments across 7 C++ (and 1 C) software projects; ConstSanitizer introduces a build slowdown of 1.05×–1.40× across all projects.

	Name	Version	Description	Build Slowdown
C++	Protobuf	2.6.1	Serialization framework	1.40×
C++	LevelDB	1.18	Key/value database	1.05×
C++	fish shell	2.2.0	UNIX shell	1.32×
C++	Mosh (mobile shell)	1.2.5	SSH replacement	1.26×
C++	LLVM TableGen	3.7.0	Domain-specific generator	—
C++	Tesseract	3.04.00	OCR engine	1.10×
C++	Ninja	1.6.0	Build system	1.20×
C	Weston	1.9.0	Wayland compositor	1.28×

indicate an incidental property of a write-through-`const`. If the code containing the writes is properly written, an object with an S/N/B/D write-through-`const` can still appear to be immutable to the client, assuming all references to that object are read-only. A write with attribute I, however, is a client-visible violation of `const`.

6 Results

We evaluated our ConstSanitizer tool on 7 C++ software projects, plus 1 C project. We attempted to choose significant benchmarks using these guidelines:

1. must span a range of application areas: applications and libraries; small, medium, and large projects; interactive and non-interactive;
2. are used by the community: the Google projects are the most popular on GitHub; the applications are popular among FOSS users; contributor-group sizes vary from a core group to a large community; and,
3. must extensively use `const` constructs.

A ConstSanitizer report indicates that a write that would not be allowed under deep immutability occurred through a read-only reference. Such writes are allowed under C++ semantics. They are only a departure from the `const` semantics that we experiment with (i.e. deep immutability with no casts and no mutable). Our experiments classify writes-through-`const` observed in actual `const`-using programs. Classifying these writes provides us valuable insight about `const` usage in practice, which will guide future work.

Our approach was to modify the project's build system to use our tool and to disable optimizations. We then ran the project's test suite, when available, and collected output from our instrumentation. Using this output we categorized the writes that we found, assigning root causes and attributes. Along with the number of static locations of writes that we found (**bolded**), we also report the number of dynamic occurrences of each write over observed executions. All else being equal, dynamic counts can help prioritize writes-through-`const`, with more-frequent locations to be investigated first. We refer to these dynamic occurrences of writes as "occurrences" in the sequel. Table 5 summarizes our benchmark projects.

We recorded relative overhead introduced by our instrumentation with respect to both building and testing times on the longest-running projects, Protobuf and LevelDB. Table 5 includes build slowdowns induced by our tool, which ranged between 1.05× and 1.40×. Our tool caused a 3.3× slowdown and 1.3× slowdown in test execution times for Protobuf and

Table 6 Protobuf shows 5 archetypes for **76** writes through `const` resulting in 127 644 occurrences.

Archetype	Locations	Occurrences	Root Cause	Attributes
Generator printer	**7**	118464	T	B & N & S
Message cache sizes	**61**	7158	M	B & S
Source code locations	**4**	1898	T	I
Linked list operations	**2**	84	M	I & S
Generate initialization method	**2**	40	M	D & N & S

Listing 5 Protobuf's Generator class performing transitive write-through-`const` to a Printer field.

```
bool Generator::Generate(...) const {
  PrintTopBoilerplate(this->printer_, ...);
}

void PrintTopBoilerplate(io::Printer* printer, ...) {
  printer->Print(...);
}
```

python_generator.cc

printer.cc

```
void Printer::Print(...) {
  WriteRaw(text + pos, i - pos + 1);
}

void Printer::WriteRaw(..., int size)
  this->buffer_ += size;
  this->buffer_size_ -= size;
}
```

transitive writes initiated by `const ::Generate()`

LevelDB respectively. The remaining projects were either interactive, or did not have long enough running test suites to get meaningful results. We do not any report LLVM TableGen numbers because we built it (with instrumentation) as part of the LLVM build process and were not able to build the executable separately.

6.1 Protobuf

Protobuf is Google's serializing framework for structured data, consisting of about 214 000 lines of C++ code. We analyzed version 2.6.1 of Protobuf by running its test suite, which contains 5 tests. Table 6 summarizes the Protobuf results. ConstSanitizer found **76** static write locations (and 127 644 occurrences). We describe 5 archetypes for these writes. An archetype is a group of writes that we judged to be similar; the writes may happen at different source locations.

The "Generator printer" archetype occurred most often. Listing 5 presents a representative expanded stack trace. The function at the top of the listing shows the initiation of the write in `Generator`'s `const`-qualified `Generate` method. This method calls `PrintTopBoilerplate`, passing a pointer to a mutable `io::Printer`. Then, `Printer`'s `WriteRaw` method modifies (root cause T) two fields: `buffer_` and `buffer_size_`. These fields are protected by a lock, act as a buffer, and are not visible outside the class (which is just a printer). This archetype also includes other `Print`-like calls with different source locations but a common explanation.

Listing 6 Protobuf's Generate initialization method performing lazy initialization.

python_generator.cc

```
bool Generator::Generate(...) const {
  this->file_ = file;
  this->printer_ = &printer;
}
```

The "Generate initialization method" archetype is related to "Generator printer". Listing 6 shows this archetype. The `printer_` field was initialized as seen above. C++ allows this write due to the `mutable` specifier. Another field, `file_`, is lazily initialized as well. Both of these fields are protected by the same lock, and are not externally visible outside the class.

We show an example of the "linked list operations" archetype in Listing 7. Here, the `depart` method grabs a lock, and uses a pointer with type `linked_ptr_internal const *`, so that the `const` applies to what is pointed to, not to the pointer. The method then modifies the `next_` field of a valid object at the point indicated by the comment. The root cause here is `mutable`: the `next_` field is declared `mutable linked_ptr_internal const* next_`. This write is a delayed initialization, not visible, and synchronized.

Listing 8 shows the "Message cache sizes" archetype. The write is protected by a lock, and is allowed by C++ because the field is `mutable`. However, this write, while involved with caching, is externally visible. The method `void SetCachedSize(int size) const` enables external code to modify this field through a `const` reference to the containing object.

Listing 9 shows the "Source code locations" archetype. The `mutable_leading_comments` method, which includes "mutable" in its name, is not declared as `const`, and thus allows writes. Its implementation writes to the `location_` field; we show an example of a caller which causes such a write. The `location_` field is externally-visible, so this is a clearly incorrect externally-visible transitive write; we assign attribute I.

We also found an archetype involving writing data to a message. This included 133 unique source locations, occurring 14 638 times in total. However, the code is heavily inlined and the build system appears to overwrite optimization settings for this subdirectory. Manual inspection of the code revealed no obvious writes. We believe this is a result of optimizations causing invalid debugging information. We thus omitted this archetype from Table 6.

6.2 LevelDB

LevelDB (1.18) is Google's lightweight key/value database library, consisting of approximately 18 000 lines of C++ code. The test suite contains 23 test drivers. There were 6 archetypes and also **6** root source locations for these writes. These locations contributed to 13 792 occurrences over the test drivers. Table 7 shows a summary of our findings for LevelDB.

Listing 10 shows the source location that caused the majority of the occurrences. This code extends the `RandomAccessFile` class to add an atomic counter field, `counter_`, that tracks the number of read calls. The root cause is that `counter_` is a pointer and is transitively written to. The reason for this write is test controllability: this class is part of the test infrastructure. Yet it must override the monitored call (and thus must be `const`). This class is meant for testing purposes only, so we concluded the write was not visible outside the class—the counter is only used in the testing code.

Listing 11 shows a modification to a caching structure that generates a new identifier. This cache is a field, `block_cache`, in `options`, which is declared as `const Options&` in `Table::Open`. The root cause is a transitive write, since the code dereferences a field of a `const` object to do the write. This write is protected by a lock and clearly involved in caching. However, it appears that other code outside of `Options` uses this block cache.

Listing 7 Protobuf using Google test linked list that writes internally.

gtest-linked_ptr.h

```
bool linked_ptr_internal::depart()
    GTEST_LOCK_EXCLUDED_(g_linked_ptr_mutex) {
  MutexLock lock(&g_linked_ptr_mutex);

  if (this->next_ == this) return true;
  linked_ptr_internal const* p = this->next_;
  while (p->next_ != this) p = p->next_;
  p->next_ = this->next_;
  return false;
}
```

Listing 8 Protobuf writing to a message's cached size field.

descriptor.pb.cc

```
int FieldDescriptorProto::ByteSize() const {
  GOOGLE_SAFE_CONCURRENT_WRITES_BEGIN();
  this->_cached_size_ = total_size;
  GOOGLE_SAFE_CONCURRENT_WRITES_END();
}
```

Listing 9 Protobuf writing to a source location object.

parser.cc

```
void Parser::LocationRecorder::AttachComments(...) const {
  this->location_->mutable_leading_comments()->swap(*leading);
}
```

descriptor.pb.h

```
std::string* SourceCodeInfo_Location::mutable_leading_comments() {
  this->leading_comments_ = new ::std::string;
}
```

Table 7 LevelDB shows writes from **6** source locations, with 13 792 occurrences in total.

Location	Occurrences	Root Cause	Attributes
db/db_test.cc:40	10311	T	N & S
util/cache.cc:315	2841	T	B & S
db/snapshot.h:54	319	T	I
db/snapshot.h:55	319	T	I
helpers/memenv/memenv.cc:274	1	T	I & S
util/testutil.h:42	1	T	N

Listing 10 LevelDB write in `db_test.cc:40` incrementing counter tracking # of writes to a file.

db_test.cc

```
class CountingFile : public RandomAccessFile {
  virtual Status Read(...) const {
    this->counter_->Increment();
  }
};
```

Listing 11 LevelDB write in `cache.cc:315` creating a new block cache in `const Options` object.

table.cc

```
Status Table::Open(const Options& options, ...) {
    rep->cache_id = (options.block_cache ?
                     options.block_cache->NewId() : 0);
}
```

cache.cc

```
virtual ShardedLRUCache::uint64_t NewId() {
  MutexLock l(&id_mutex_);
  return ++(last_id_);
}
```

Listing 12 LevelDB write in `snapshot.h` deleting a list element and updates pointers.

snapshot.h

```
void Delete(const SnapshotImpl* s) {
  assert(s->list_ == this);
  s->prev_->next_ = s->next_;
  s->next_->prev_ = s->prev_;
  delete s;
}
```

Listing 12 shows a modification of a linked list node accessed through two pointer dereferences. This corresponds to both `snapshot.h` locations shown in the table. This code modifies the pointers obtained from following its own nodes, performing a transitive write through a `const` qualifier. We do not know why the developers declared `s` as `const` since it is also destroyed at the end of the method. In any case, we assigned attribute I.

Listing 13 shows a write-through-`const` in the `InMemoryEnv` class. As with the cache, the root cause is a transitive write: in the caller, `options` is declared `const Options&`. Unlike the caching example, this file isn't involved in caching and appears to be a visible change to `options`. This write is protected by a lock, giving attribute I & S.

Listing 14 shows the final write-through-`const`-qualifier that we found for LevelDB. The caller location is the same as in Listing 13 above. In this case, however, the containing class extends `InMemoryEnv` and adds a field to count the number of errors (for testing purposes only). Therefore we attribute this write as being not visible—it is only used in tests.

6.3 fish shell

fish shell (2.2.0) is a UNIX shell providing advanced features, consisting of approximately 48 000 lines of C++ code. We compiled the project with our tool and executed an instance of the shell. Our workload launched the shell and immediately exited. We found writes from **4** unique source locations for 98 occurrences in total. All locations are within the `exchange` function. Listing 15 shows this function along with a snippet of `_wgetopt_internal` that calls `exchange`. The root cause is that the `const`-qualified `argv` variable gets cast to non-`const` and then passed to `exchange`. This write shows that the `const`-qualifier on `argv` is incorrect and should not be included.

6.4 Mosh (mobile shell)

Mosh (mobile shell) (1.2.5) is a remote terminal application that is a replacement for secure shell (SSH), consisting of about 13 000 lines of C++ code. Our workload was to launch the mosh server and immediately terminate it. We found writes-through-`const` at **8** unique

Listing 13 LevelDB write in `memenv.cc` changing the environment in options object.

```
Status DB::Open(const Options& options, ...) {                db_impl.cc
  s = options.env->NewWritableFile(...);
}

... InMemoryEnv::NewWritableFile(...) {                       memenv.cc
  MutexLock lock(&mutex_);
  this->file_map_[fname] = file;
}
```

source locations (432 occurrences). Listing 16 shows one of the writes. Mosh parsing code sets a flag to indicate completion. However, the developers declared the parser action as `const` in the same method where they modify it. The root cause is that the variable `handled` is declared `public mutable`. We believe this is an incorrectly `const` qualified variable.

6.5 LLVM TableGen

We instrumented LLVM's (3.7) TableGen executable, which uses domain-specific information to generate files with custom backends. This part of LLVM consists of approximately 34 400 lines of C++ code. It is primarily used in building LLVM itself. We added our instrumentation to the build system and observed reports from an instrumented version of TableGen executing as part of the build process. LLVM itself is a large body of code with too many writes-through-`const`-qualifier objects to manually classify. In TableGen, we found writes from **3** unique source locations (282 occurrences).

The handling code for DFAs contains some puzzling writes, shown in Listing 17. The write immediately follows an instantiation of a `const State` object. The `State` class itself is only available in a file's translation unit (not usable outside the file), which may indicate that the `State` is not intended to be widely used. `State` only contains `const` methods and all of its fields (except one explicitly declared `const`) are mutable. Since all methods are `const` there is no difference in callable methods between non-`const` and `const`-qualified access. In addition, since all other fields are mutable, developers are allowed to re-assign the same fields in a `const` method as they would in a non-`const` method. Since only one field doesn't have `mutable`, developers could achieve the same effect by making all methods non-`const`, removing all `mutable` specifiers on fields, and changing the one field that did not have `mutable` to be `const` qualified.

The other write is in the code that computes a sub-register index for code generation. Listing 18 shows the containing method. The root cause here is that the `LaneMask` field is mutable. The write caches the value. However, this value is not used in any other methods.

6.6 Tesseract

Tesseract (3.04.00) is an optical character recognition (OCR) engine maintained by Google, consisting of 147 000 lines of C++ code. This project does not contain any easy-to-run tests. We compiled it with our tool and ran it with invalid arguments. With our limited knowledge of Tesseract's usage, we were not able to cause the core algorithm to execute. However, we found a strange write, shown in Listing 19. The root cause is that the `used_` field is mutable. This write appears to be an incorrect usage of `const`. Strangely, however, the comments indicate that is a defensive write against possible further writes-through-`const`-qualifiers.

Listing 14 LevelDB write in `testutil.h` injecting faults into the test suite.

testutil.h

```
... EnvError::NewWritableFile(...) {
  ++this->num_writable_file_errors_;
}
```

Listing 15 fish shell writing to `const`-qualified `argv` object.

wgetopt.cpp

```
... _wgetopt_internal(..., wchar_t *const *argv, ...) {
    exchange((wchar_t **) argv);
}

... exchange(wchar_t **argv) {
    argv[bottom + i] = argv[top - (middle - bottom) + i];
    argv[top - (middle - bottom) + i] = tem;
    argv[bottom + i] = argv[middle + i];
    argv[middle + i] = tem;
}
```

Listing 16 Mosh handling terminal action with a write-through-`const`.

terminal.cc

```
void Emulator::print(const Parser::Print *act) {
  act->handled = true;
}
```

Listing 17 LLVM DFA code marks `State const` for no apparent reason.

DFAPacketizerEmitter.cpp

```
void DFAPacketizerEmitter::run(raw_ostream &OS) {
  const State *NewState;
  NewState = &D.newState();
  NewState->stateInfo = NewStateResources;
}
```

Listing 18 LLVM SubReg writes to a `mutable` field in a `const` method.

CodeGenRegisters.cpp

```
unsigned CodeGenSubRegIndex::computeLaneMask() const {
  if (this->LaneMask)
    return this->LaneMask;
  this->LaneMask = ~0u;
  unsigned M = ...;
  this->LaneMask = M;
  return this->LaneMask;
}
```

Listing 19 Tesseract performs a strange write in its string class.

strngs.cpp

```
const char* STRING::string() const {
  const STRING_HEADER* header = GetHeader();
  header->used_ = -1;
  return GetCStr();
}
```

```
/* mark header length unreliable
because tesseract might cast away
the const and mutate the string
directly. */
```

Listing 20 Ninja write-through-`const` in test code.

```
TimeStamp StatTest::Stat(const string& path, ...) const {
  this->stats_.push_back(path);
}
```
disk_interface_test.cc

Listing 21 Weston option parser modifying its `const` option argument.

```
handle_option(const struct weston_option *option, ...) {
  * (char **) option->data = strdup(value);
}
```
option-parser.c

6.7 Ninja

Ninja (1.6.0) is a build system consisting of approximately 14 900 lines of C++ code. It includes a modest test suite. Our tool reports 39 occurrences from calls to the standard library. All of these warnings have a **single** source location outside the standard library: `src/disk_interface_test.cc:226:3`. Listing 20 shows this static source location. This is a quick hack to run the test suite with the same API as normal clients. This field stores statistics that are checked in the test suite only. The field is mutable and not seen outside the test suite, so we give this write the "not visible" attribute.

6.8 Weston

While we focus on C++ in this work, our technique also works on `const` in C programs. We therefore evaluated it on a C application. Since this is the sole C project, we omit Weston from the overall table of results (Table 8). Weston (1.9.0) is a reference implementation of a Wayland compositor. It consists of approximately 85 000 lines of C code and the test suite has 20 tests. We did not expect to see many writes, as most C standard library functions do not require `const` (corresponding C++ library functions usually do), and also due to annoyances in using `const` in C, which we describe below. However, even with a small test suite, we found **4** unique source locations for writes (accounting for 115 occurrences).

All of the writes-through-`const` are transitive and came from parsing code. The argument option parser accounts for **3** locations. Listing 21 shows a write in the parser. Function `handle_option` does not modify the pointer value of `option` but modifies its transitive data field. This does not change any data stored in the `weston_option` structure, maintaining bitwise `const`-ness. The `data` field's type is `void *` and the cast does not remove `const`. Based on the function name, one might expect a write to the `data` field, not its pointee.

The final location was in the configuration file parsing code. Listing 22 shows the `weston_config_section_get_uint` function dereferencing and modifying the `value` argument passed in from a field of a `const` struct. As above, based on the function naming, one would expect any writes to happen through the `dest` pointer. This write does not modify any data stored in the `config_command` structure and maintains bitwise `const`-ness as well.

We made an observation as to why `const` may be unattractive to C developers: there is no clean way to initialize a structure analogous to C++ constructors/destructors. A popular C idiom is to assign constructor-like functions signatures like `rec_init(struct rec *r)`. This signature prevents initialization without casting: `const struct rec r; rec_init(&r)` is illegal. However, it is cumbersome to always cast for constructor-like calls. One could change the signature of the function to `rec_init(const struct rec *r)` and perform the cast in the function. However, that function would violate shallow immutability—it writes

Listing 22 Weston config parser writing to its `value` argument.

```
struct config_command {
  char *key;
  uint32_t *dest;
};
const struct config_command *command = ...;
```

hmi-controller.c

```
weston_config_section_get_uint(..., command->dest, ...);
```

```
... weston_config_section_get_uint(..., uint32_t *value, ...) {
    *value = strtoul(entry->value, &end, 0);
}
```

Table 8 Writes-through-`const`-qualifiers in our other benchmark programs were mainly incorrect uses of `const`.

Project	Location	Occurrences	Root Cause	Attributes
fish shell	`wgetopt.cpp`	98	C	I
Mosh	`terminal.cc`	432	M	I
LLVM	`DFAPacketizerEmiter.cpp`	112	M	I
TableGen	`CodeGenRegisters.cpp`	170	M	B & N
Tesseract	`ccutil/strngs.cpp`	1	M	I
Ninja	`disk_interface_test.cpp`	39	M	N

to fields as it initializes them. Using `const` in C appears to require developers to ignore the casting away of `const` qualifiers for constructor-like functions.

6.9 Summary

Table 8 summarizes the writes-through-`const`-qualifiers from benchmarks other than Protobuf and LevelDB. Across the 7 C++ projects we instrumented and ran, we observed 17 unique archetypes across a total of 142 288 dynamic occurrences. We manually divided these archetypes into 17 classifications. The root causes were evenly split between writes through mutable fields and transitive writes (8 of each) with one write-through-`const` due to casting. Valid attributes were mostly with-synchronization and because the write was not visible (7 and 6 respectively). The other valid attributes, writing to a buffer/cache and delayed initialization, occurred 4 times and 1 time respectively. The majority attribute, in 9 cases, was that the write was incorrect and violated intuitive notions of what `const` should mean. We reported our results to developers. Within a few days the developers simply removed incorrect `const` qualifiers in both fish and Mosh.

We found 3 projects (LevelDB, Mosh, and Ninja) had writes that only occurred in tests. The writes-through-`const` we found were in testing code; writes were to counters only present in test environments. In Mosh, the fact that the writes were only for test purposes was not immediately obvious. However, discussions with the developers revealed that the `handled` variable was only used for debugging. All of these writes-through-`const` are related to test controllability, suggesting that this idiom should be supported directly in the programming language.

7 Related Work

We discuss a number of areas of related work: immutability (and related approaches) for Java; purity analyses; and dynamic analyses that inspired our approach. The Java programming language has no exact analog to C++'s `const` operator. Related work defined immutability annotations for Java and statically and dynamically verified that programs satisfy their annotations. Potanin et al [9] provide a recent discussion of immutability terminology and compare research implementations in depth. (Our implementation has at its core a dynamic analysis verifying that C++ programs satisfy a strengthened version of their `const` annotations.) A different stream of related work verifies whether (Java) methods are pure, i.e. have no visible side-effects; there is a strong connection between purity and immutability.

`const` for Java, and similar projects. Javari [17, 18] allows its users to specify read-only references in Java programs. It aims to ensure that a `readonly` (effectively, deeply-immutable `const`) typed object does not mutate its state or any state transitively reachable through its references. Javari, like C++, includes a `mutable` keyword, which allows developers to specify that a field may be modified regardless of `readonly` qualifiers. Javari also inserts dynamic checks to verify that downcasts maintain the immutability qualifier of the type. Essentially, Javari provides a safer version of C++'s `const` that, due to the nature of Java, maintains deep immutability unless the developer explicitly opts out of the checks.

Our work provides similar guarantees to those provided by Javari. Since Javari does not have an underlying C++ `const` specification to build on top of, it has to implement all of those checks itself. In terms of what we check, our work matches Javari in terms of transitivity and downcasts. Our work also reports writes to mutable fields at runtime.

Unlike Javari, we could investigate the behaviour of a set of real-world benchmark programs, developed against the C++ `const` semantics (and which, by necessity, satisfy those semantics). Our empirical study therefore points out the difference in practice between `const` semantics as they exist—shallow immutability, mutable, and `const` casting; and a stronger version of these semantics—deep immutability, no mutable, and no const casting.

A related concept to `const` is that of stationary fields, as proposed by Unkel and Lam [19]. A stationary field is similar to a Java `final` field. A Java `final` field can only be written to in a constructor. By contrast, a stationary field is only written to before it is read. In essence, a stationary field acts like a `final` field but with fewer restrictions on where initialization may occur. Nelson et al [8] performed a follow-up study using a dynamic analysis which determined that 72–82% of fields in Java programs are stationary. Their work, like ours, empirically explores how programs use (or could use, in their case) language features.

Purity analyses. A pure method does not modify any state accessible before the method was called. Pure methods may create and modify objects to return to the caller. A function which writes to no global state and has all arguments transitively `const` is pure.

ReIm and its corresponding type inference system, ReImInfer [5], is similar to Javari, except that its type system is context sensitive. ReIm was designed to find method purity. Its type system allows the immutability of the return type to match that of the calling reference. This allows methods to be reused without requiring mutable and read-only versions. ReImInfer is a type inference system that maximizes the amount of methods marked as `readonly`. They report that 41–69% of methods can be marked as `readonly`.

Other systems also verify method purity. JPPA is a combined pointer and escape analysis for Java [12, 11]. Sălcianu and Rinard found over half the methods they analyzed were pure. Rytz et al [10] instead use a simpler flow-insensitive analysis and find similar results.

Artzi et al proposed a combined dynamic and static analysis for mutability [1]. Their analysis determines the mutability of method parameters. The goal of that work was to scale better and produce more precise results than static analysis alone.

`const`-correctness through abstract machines. Chisnall et al [3] propose a memory-safe abstract machine for C which can be used to verify that immutable objects (declared with `const`) are never mutated through non-`const` aliases. This resembles our approach of using shadow values to track the declared `const`-ness of an object. (They do not investigate transitive immutability.) Our study, by contrast, focuses solely on writes-through-`const`. All of their subject programs removed a `const` qualifier at some point. It was beyond the scope of their work to investigate how and why their subject programs removed the `const` qualifier.

Other dynamic analyses. Our dynamic analysis approach is similar to approaches used in Umbra [20] and Dr. Memory [2]. Like Umbra and Dr. Memory, we use shadow values to detect interesting program behaviours. However, ConstSanitizer builds directly on LLVM and does not use a dynamic instrumentation platform. Furthermore, the properties that we verify are novel `const`-related properties, compared to Dr. Memory, which looks for memory errors such as accesses to unallocated space, and Umbra, which helps developers understand threads' memory access patterns and implements almost-free custom watchpoints.

8 Conclusion

The `const` qualifier in C++ is extensively used in real-world code, but developer intent behind `const` usage is unclear. In this paper, we have presented our ConstSanitizer system. ConstSanitizer dynamically detects writes through `const` qualifiers which are legal in C++ but which modify state transitively starting from a `const` qualifier, write to mutable fields, or write to values whose `const`-ness has been cast away. Our results show that, although writes through `const` are ubiquitous across our 7 C++ and 1 C benchmark programs, there are only a small number (17) of archetypes these writes. We used our results to develop a classification of writes according to root cause (transitivity, mutable field, or const cast) and attributes (synchronized, not visible, buffer/cache, delayed initialization, incorrect). Our work helps understand how the `const` qualifier is used in practice and leads us to conclude:

- Developers definitely violate bitwise `const` (**RQ1**).
- The majority of write-through-`const` archetypes (9/17) are incorrect code which observably change an object's state.
- On our benchmarks, programs write-through-`const` about equally often using transitive writes through fields (8/17) and writes to mutable fields (8/17) (**RQ2**).
- We observed four classes (N, B, D, S, discussed in Section 5) of valid reasons for writes-through-`const`-qualifiers. For instance, sometimes developers write-through-`const` to delay initialization or implement buffer caches. Such writes should be automatically validated by yet-to-be-developed tools.
- About half (9/17) of the observed usages are invalid, consisting of methods which implemented exceptions to an object's const-ness; perhaps the developers chose to add one exception rather than remove const completely. (**RQ3**)

References

1 Shay Artzi, Adam Kiezun, David Glasser, and Michael D. Ernst. Combined static and dynamic mutability analysis. In *ASE*, pages 104–113, November 2007.

2 Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *CC*, pages 213–223, 2011.

3 David Chisnall, Colin Rothwell, Robert N. M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the PDP-11: architectural support for a memory-safe C abstract machine. In *ASPLOS*, 2015.

4 Felix Fang. Personal communication, 2015.

5 Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *OOPSLA*, 2012.

6 ISO. Programming languages—C++. N3690, May 2013.

7 Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs.* Addison Wesley, 3rd edition, 2005.

8 Stephen Nelson, David J. Pearce, and James Noble. Profiling field initialisation in Java. In *RV*, volume 7687 of *LNCS*, pages 292–307, 2012. `doi:10.1007/978-3-642-35632-2_28`.

9 Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. Immutability. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *LNCS*, pages 233–269. 2013.

10 Lukas Rytz, Nada Amin, and Martin Odersky. A flow-insensitive, modular effect system for purity. In *FTFJP*, July 2013.

11 Alexandru Salcianu. *Pointer Analysis for Java Programs: Novel Techniques and Applications.* PhD thesis, MIT, 2006.

12 Alexandru Salcianu and Martin C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, pages 199–215, January 2005.

13 Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

14 Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *CGO*, pages 46–55, 2015.

15 Herb Sutter. GotW #6a solution: Const-correctness, part 1. `http://herbsutter.com/2013/05/24/gotw-6a-const-correctness-part-1-3/`, May 2013. Accessed Dec 2015.

16 LLVM Team. The LLVM compiler infrastructure. `http://llvm.org/`, December 2015.

17 Matthew S. Tschantz. Javari: Adding reference immutability to Java. Master's thesis, Massachusetts Institute of Technology, 2006.

18 Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, October 2005.

19 Christopher Unkel and Monica S. Lam. Automatic inference of stationary fields: a generalization of Java's final fields. In *POPL*, pages 183–195, January 2008.

20 Qin Zhao, Derek Bruening, and Saman P. Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *CC*, pages 22–31, 2010.

LJGS: Gradual Security Types for Object-Oriented Languages

Luminous Fennell[1] and Peter Thiemann[2]

1 **University of Freiburg**
Georeges-Köhler Allee 79, 79110 Freiburg, Germany
fennell@informatik.uni-freiburg.de

2 **University of Freiburg**
Georeges-Köhler Allee 79, 79110 Freiburg, Germany
thiemann@informatik.uni-freiburg.de

Abstract

LJGS is a lightweight Java core calculus with a gradual security type system. The calculus guarantees secure information flow for sequential, class-based, typed object-oriented programming with mutable objects and virtual method calls. An LJGS program is composed of fragments that are checked either statically or dynamically. Statically checked fragments adhere to a security type system so that they incur no run-time penalty whereas dynamically checked fragments rely on run-time security labels. The programmer marks the boundaries between static and dynamic checking with casts so that it is always clear whether a program fragment requires run-time checks. LJGS requires security annotations on fields and methods. A field annotation either specifies a fixed static security level or it prescribes dynamic checking. A method annotation specifies a constrained polymorphic security signature. The types of local variables in method bodies are analyzed flow-sensitively and require no annotation. The dynamic checking of fields relies on a static points-to analysis to approximate implicit flows. We prove type soundness and non-interference for LJGS.

1998 ACM Subject Classification D4.6. Security and Protection

Keywords and phrases gradual typing, security typing, Java, hybrid information flow control

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.9

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.2.1.4

1 Introduction

Information-flow control (IFC) is a cornerstone of language-based security. A typical IFC policy rules out the flow of information from classified sources to public sinks. The technical property aimed for is noninterference: changing a classified source does not influence the public sinks. Noninterference comes in different flavors depending on the observational capabilities of an attacker (e.g., termination sensitive or not, batch or interactive).

There are also different kinds of IFC. A static system performs a static analysis, for instance using a security type system, and guarantees noninterference for analyzed programs. A dynamic system attaches run-time security labels to values, propagates them along with the values, and checks them at appropriate points during program execution. Hybrid systems employ additional static analysis to improve precision in the detection of implicit flows [23].

The choice between a static and a dynamic system is a difficult trade-off in the development of secure software. For new software components that are designed with security in mind

and for components that have high reliability requirements a static analysis is a good fit: It does not impose any run-time overhead and provides hard static guarantees whereas a dynamic system needs to manage security labels at run-time and often aborts a running program when a security problem is detected.

For the integration of legacy components as well as for prototyping, static IFC can impede programming productivity significantly: Simple analyses yield many false positives and thus may reject many programs that are in fact secure. Precise analyses often require extensive manual annotation of the source code. They rely on complex abstract domains that may be computationally expensive or difficult to understand by non-experts. In contrast, dynamic approaches rely on an instrumented run-time system. Thus, after specifying a security policy, a programmer can debug policy breaches by testing and inspecting concrete program runs.

Contributions

To make the benefits of both approaches available to mainstream object-oriented programming, we propose a type-based amalgamation of static and dynamic IFC for Java. Inspired by work on gradual typing [26, 14], our system enables programmers to rely on the guarantees and efficiency of a lightweight security type system and to back off to dynamic checking when necessary. Gradual typing enables the composition of programs from typed and untyped fragments using suitable type casts at the interfaces. It guarantees full compliance with the type system in the typed fragments, whereas untyped fragments may raise run-time type errors. *Gradual security typing* transposes this approach to an information flow setting.

Lightweight Java with Gradual Security (LJGS) is an object-oriented core calculus that implements our ideas. LJGS is a subset of Java without threads, exceptions, or reflection. It is inspired by Lightweight Java (LJ) [29]. Our specification of LJGS is type-based: it takes the form of a type system and it assumes that the underlying program is well-typed. Thus, LJGS is independent of specific Java typing features like generics. The LJGS types in this paper amount to type annotations in an actual Java program.

Following Denning [10], LJGS specifies permissible information flows using a lattice $(Sec, \sqsubseteq)$ of *security levels*. For each field of a class, the programmer specifies a fixed security level *or* marks the field as dynamically checked. For each method, the programmer supplies a constrained polymorphic security type signature (see examples in Section 2). The signature relates the security types of the arguments, the result, and the effect on global variables with each other and with fixed levels that arise from field accesses. The security types of local variables need *not* be declared. Moreover, the types of local variables are *flow sensitive*, that is they can change during the method; object field types are *flow insensitive* to keep static checking light-weight. In contrast to other work on security type signatures for Java [30] our type checking algorithm works directly on the constraints.

The LJGS type system is designed so that the security level of a value with a static security type does not have to be tracked at run-time. Each value with a dynamic security type, in a dynamic field or in a temporarily dynamic local variable, carries a run-time security label that overapproximates its true security level. Implicit flows are detected with an approach inspired by *hybrid monitors* [23]: at the join points in the control-flow graph, security labels of variables and fields are upgraded preemptively if they could be updated in the untaken branches. A points-to analysis assists in determining the heap references to fields that require upgrading. Purely dynamic approaches like the no-sensitive-upgrade (NSU) policy [3, 36] may be used as well, but with more conservative results.[1]

[1] We used NSU in a previous version of this paper: `http://proglang.informatik.uni-freiburg.de/projects/gradual/ljgs/fcs2015.pdf`

No existing system provides a similarly powerful combination of static and dynamic IFC for a Java-like language. In particular, prior work [12, 13] illustrates the principles of gradual security typing with toy languages. While LJGS is the result of extrapolating similar principles to an object-oriented setting, extensions required in practice, like polymorphic signatures and avoiding run-time checks for static code, yield a type- and run-time system that differs significantly from prior designs. The differences are further elaborated in Section 9.

Technical results: Besides proving type preservation and progress (up to breaches of the security policy in dynamic updates), our key result is *batch termination insensitive non-interference* (BTINI), as defined by Askarov et al [2]. We assume that an attacker can only access the public part of the heap, may construct the public arguments of any LJGS method, run it, and inspect only the public part of the result and the heap afterwards. If the method diverges or aborts, the attacker receives no information. There is no way to obtain intermediate results.

Scope of the LJGS calculus

LJGS supports the features of class-based, typed object-oriented languages that pose fundamental challenges for information flow control: mutually recursive methods and classes, mutable local variables, mutable objects, and virtual method calls. In the following, we summarize these challenges and then discuss features omitted from LJGS, namely exceptions, and Java-style downcasts and type-tests. Furthermore, Section 7 discusses how a practical language based on LJGS could deal with reflective and otherwise hard-to-analyze code. **Mutually recursive classes and methods** complicate global, inter-procedural inference of types and information flow. LJGS supports local type inference, only. It avoids the need for global inference with annotations on methods and classes. The issue of **mutable local variables** arises already in simple imperative languages [24], but it is not addressed by previous work on gradual security [12, 13]. For LJGS, we adapt best practice in maintaining a *program counter type* to track implicit flows and in handling variables flow-sensitively [17, 23] to improve precision.

Mutable objects may lead to *global* implicit flows, as the following example shows:

```
class C { int F; void setF(){ this.F = 1; };}
C c = new C(); C d = c;
if (secret == 42) {d.setF();}
printPublic(c.F); // <- information leak
```

Here, `secret` has a high-security type and `printPublic` is a low-security sink. The field `F` of the object `c` is updated under a high-security program counter by calling method `setF()` on `d` which is an alias of `c`. Afterwards, `c.F` is publicly exposed by `printPublic()`. However, there is no local indication for the leak: the program does not directly change `c.F`. The reason for the leak is that `c`, `d`, and **this** in the method call to `setF` are aliases for the same object during the execution of the program. Sound information flow control has to prevent low-security access to the field `c.F` after the update, even if the update is performed on a different alias or in a different method.

In LJGS and other security-typed languages [21, 22] this requirement is enforced by fixed, flow-insensitive security types for fields and by write-effect annotations on methods that indicate the types of fields that a method updates. For static, high-security program counters, low-security or dynamic effects are forbidden. Dynamic program counters admit only dynamic effects and dynamic fields updates. LJGS treats mutable objects differently

than previous work[13], where run-time labels are required even for statically typed object references.

Virtual method calls cause conditional control flow and thus may create implicit information flows: the run-time type of the receiver object selects the implementation of a method to execute. While dynamic enforcement of information flow copes naturally with dynamic dispatch, statically checking a virtual method requires a sound, static approximation of the security properties of all possible implementations that might be executed. LJGS supports dynamic dispatch for Java-like class derivation and method overriding. To soundly type-check method calls, the type system imposes a restriction on signatures of overriding methods (cf. Definition 5, Section 4.4).

Among the features that LJGS omits, support for (catchable) **exceptions** is the most challenging for information flow control:[2] throwing an exception introduces non-local implicit information flows from the program counter of the throw-site to any exception handler in the call stack. Practical languages with type-based support for information flow control like JIF [21] and FlowCaML [22] solve this problem with additional effect annotations on methods that track the program counter type at the throw-site. This exception effect is an appropriate lower bound for the program counter type of exception handlers. To simplify the presentation in this paper, we do not consider catchable exceptions, but we expect the extension of LJGS's type system with exception effects to be unsurprising, as the typing constraints of LJGS are similar to FlowCaML's constrained type schemes.

Banerjee and Naumann identify (down-)**casting of classes and type test** as potential sources of information flow [6]. A conditional based on a type test creates an information flow from the tested object to the program counter of the branches. For example, the body of the condition in line 5 executes with a high-security program counter, as the run-time class of `c` depends on the value of the high-security variable `secret` (line 4).

```
1  class C { }
2  class D extends C { }
3  C c;
4  if (secret == 42) { c = new D(); } else { c = new C(); }
5  if (c instanceof D) {
6   ... // <-- high-security program counter
7  }
```

LJGS does not support type tests directly, but as the objects involved in branch conditions are already tracked, the feature would be straightforward to add as a primitive operation. Downcasts may result in additional exceptional flows when a cast error is recoverable. As we currently do not consider exceptions for LJGS, we also omit downcasts for simplicity.

2 A Taste of LJGS

This section demonstrates the salient features of LJGS. The security lattice used in the examples has at least two points, `LOW` and `HIGH`, where $\texttt{LOW} \sqsubseteq \texttt{HIGH}$ and $\texttt{HIGH} \not\sqsubseteq \texttt{LOW}$.

Figure 1 illustrates how to specify information flow policies via security signatures. The method `max` calculates the maximum of its parameters `x` and `y`. Clearly, the method's result depends on both parameters. To express this dependency, method `max` carries polymorphic constraints in the **where** clause of its signature. It indicates that information may flow from `x` and `y` to the result by asserting that the parameters' security types are lower bounds for the type of the return value. The variables `x`, `y`, and `ret` stand for the security type of

[2] It is unproblematic to extend LJGS with *uncatchable* exceptions, as abrupt program termination does not violate BTINI.

```
1  int max(int x, int y) where { x ≤ ret, y ≤ ret } {
2    if (x ≤ y) {x = y;} return x;
3  }
4
5  class Logger { String[LOW] buf; String[HIGH] hbuf; String[⋆] dbuf; }
6
7  int maxMsg(Logger log, int x, int y)
8    where { x ≤ ret, y ≤ ret , log ≤ LOW} and { LOW } {
9    if (x ≤ y) { x = y; }
10   log.buf = "max was called";
11   return x;
12 }
```

Figure 1 Examples of method definitions and polymorphic signatures.

```
1  Logger log = new Logger(); int x = (⋆ ⇐ HIGH) secret;  int y = (⋆ ⇐ LOW) 42;  int r;
2
3  r = this.maxMsg(log, secret, 42);
4  r = this.maxMsg(log, x, y);
5
6  // type error: uncontrolled mix of dynamic and static arguments
7  // r = this.maxMsg(log, secret, y);
8
9  if (x == 42) {
10   // type error: program counter is statically unknown
11   // r = this.maxMsg(log, x, y);
12 }
```

Figure 2 Calling polymorphic methods.

the parameters `x`, `y`, and the return value. As the variables are not bounded by concrete security levels, the signature is polymorphic and `max` may be called under any program counter type with arguments that satisfy the constraints. It may also be called with two dynamic arguments.

Line 5 of Figure 1 defines a class `Logger` with three fields: `buf` has with static security type `LOW`, `hbuf` with static security type `HIGH`, and `dbuf` with dynamic type $\star$. The method `maxMsg` is a version of `max` with global side effects; it writes to the low-security field of a `Logger` object. The signature of the method indicates this global effect to a `LOW` field in the **and** part of the **where** clause. LJGS requires that the *program counter type* at all call sites is smaller than the declared effect `{LOW}`, which guarantees that `maxMsg` is only called securely in contexts where the program counter security level is `LOW`. The signature of `maxMsg` additionally enforces that the `log` parameter is a low-security reference by upper-bounding it with `LOW`. The parameters `x` and `y` remain polymorphic, as they do not influence the global side effect. Typically, parameters have concrete security types as upper bound, like `log` in method `maxMsg`, if either the method requires *read or write access* to a field of that type or if the parameter flows into a program counter for a field update.

Figure 2 illustrates the use of polymorphic methods in dynamic and static code. Dynamically checked values are created with a *value cast* like `(⋆ ⇐ HIGH)secret`. The two casts in line 1 create a dynamic version of the high-security value in variable `secret` and a low-security constant `42`, storing them in `x` and `y`, respectively. The *source type* of a cast, here `HIGH` and `LOW`, respectively, indicates what run-time label to attach to the cast value. The *destination type* is the dynamic type, $\star$. The polymorphic method `maxMsg` may be called with arguments that are either all static (line 3) or all dynamic (4). The recipient `r` is typed flow-sensitively and has type `HIGH` in line 3 and type $\star$ in line 4.

Calling `maxMsg` with mixed static and dynamic arguments, as shown in line 7, is not allowed in this example: The dynamics of LJGS (cf. Sections 5 and 6) do not represent static

```
1  int maxMsgDyn(Logger log, int x, int y)
2      where { x ≤ ret, y ≤ ret, log ≤ ⋆} and { ⋆ } {
3    if (x ≤ y) { x = y; }
4    log.dbuf = "max was called"; return x; }
5
6  void logResults(Logger log, boolean privMode, int h, int l)
7    where { HIGH ≤ h , l ≤ LOW,
8            , privMode ≤ ⋆, privMode ≤ LOW , log ≤ ⋆, log ≤ LOW}
9     and { ⋆, LOW } {
10   log.dbuf = "public result:" + (⋆ ⇐ LOW) l;
11   if (privMode) {log.dbuf += "secret result:" + (⋆ ⇐ HIGH) h;  }
12   /* ... */
13   if (!privMode) {log.buf = (LOW ⇐ ⋆) log.dbuf;  }
14 }
15
16 boolean high = ...; // a high-sec. value
17 maxMsgDyn(log, x ,y);
18 if (high) {(HIGH ⇛ ⋆) {  maxMsgDyn(log, x ,y); }}
```

Figure 3 Examples of methods using dynamic IFC.

security information at run-time. Thus, information flow between static and dynamic types cannot be controlled only by inspecting run-time security labels and the type system forbids flows from static to dynamic entities except when they are explicitly controlled by casts. Uncontrolled flows from dynamic to static entities are also forbidden, as dynamic labels are not available at type-checking time. To enforce both restrictions, constraints of the form $x \leq y$ are not satisfiable if x is static and y is dynamic, or vice-versa. Thus the call in Line 7 violates the signature of `maxMsg` which relates both arguments with **ret**.

LJGS imposes similar restrictions for implicit flows. For example, the call to `maxMsg` in line 11 would be rejected. Testing the dynamic variable creates a dynamic program counter that cannot be checked against the static effect of `maxMsg`'s signature.

Figure 3 illustrates the distinguishing feature of LJGS: the integration of dynamic and static enforcement of security policies.

The method `maxMsgDyn` is a less restrictive version of `maxMsg` that relies on dynamic IFC. As before, its signature states that `x` and `y` flow into the return value. But instead of a `LOW` effect, it asserts a dynamic effect and requires that the `log` object admits dynamic access (i.e., is smaller that $\star$). While `maxMsg` writes to a low-security field and is thus incompatible with high-security program counters, `maxMsgDyn` uses the dynamic field `dbuf`. Dynamic fields are checked flow-sensitively at run-time, so that LJGS permits calls to `maxMsgDyn` in low- and high-security program counters (lines 16 to 18 in Figure 3). After line 17, `log.dbuf` contains a dynamic low-security string. The statement starting with "`(HIGH` $\Rrightarrow\star$`)`" in line 18 is a *context cast* which instructs the run-time system to propagate a high-security program counter label for the statements contained in the cast. As a result, the dynamic label of `log.dbuf` can be updated to a high-security label after the call in line 18 to reflect the implicit flow from the program counter into `log.dbuf`.

The method `logResults` (also in Figure 3) shows how to adapt information flow behavior dynamically using a boolean flag. It takes as arguments a `log` object, a boolean flag `privMode` and two integers `h` and `l`. The method's purpose is to write the values of `l` and `h` to various buffers in the `log` object, depending on the flag `privMode` ("*private mode*"). The signature states that `h` is `HIGH`[3] and `l` is `LOW`. It also states that the method has both, an effect of type dynamic ($\star$) *and* static-low (`LOW`). Similarly it requires `log` to admit accesses to fields of type

[3] A concrete security type as lower bound to a parameter is never required in LJGS. Here it is specified to make `h` a high-security value for the sake of the example.

```
1  interface Modifier { int modify(int x) where { x ≤ ret }; }
2
3  int maxMod(int x , int y, Modifier ymod) where { x ≤ ret, y ≤ ret, ymod ≤ ret }{
4     z = ymod.modify(y);
5     if (x ≤ z) { return z } else { return x }
6  }
```

Figure 4 Example of object-oriented code.

$\star$ and `LOW`, and that `privMode` can flow into fields of type $\star$ and `LOW`. The reason for these constraints become clear when inspecting the method's body: in line 11 there is a dynamic write of a dynamic high-security value to `dbuf` in the context of `privMode` and `log` and in line 13 there is a static write to `buf` using the value of `dbuf` cast to `LOW` in the context of `privMode` and `log`. Despite the copy of the potentially high-security content of `dbuf` to `buf` in line 13, the method is actually secure: the flag `privMode` guards the field updates such that a high-security content of `dbuf` is never written to `buf`. If the author of `logResults` would have failed to guard the field updates correctly, for example by accidentally omitting the negation of `privMode` in line 13, the value cast from $\star$ to `LOW` would have aborted the program by signalling a dynamic security violation. Using boolean flags in such a way is only possible in the dynamic fragment of LJGS; if `log.dbuf` were static, the type system would require it to have a high-security type and unconditionally forbid the copy to `log.buf`.

It remains to explain how the constraints for `privMode` and `log` can be possibly satisfied, given the premise that a value cannot be implicitly converted between static and dynamic types. LJGS includes a special bottom type, called the *public type* ($\bullet$), for this purpose. The type systems ensures that such a public value carries no confidential information so that it can be used in dynamic code without carrying a run-time label.

Figure 4 shows a final example that illustrates the interaction of LJGS' gradual security typing with object-oriented code. The interface `Modifier` specifies a method `modify` with a signature that asserts no global side effects and where the input flows into the output. The method `maxMod` calculates the maximum of its arguments `x` and `y` after modifying `y` using the `Modifier` object `ymod`. Its signature contains the same constraints as `max` in Figure 1 and adds the additional requirement that `ymod` is less confidential than the result.

The following code snippet implements the polymorphic `max` method with `maxMod`:

```
class Id extends Modifier {int modify(int x) where { x ≤ ret } { return x; }}
// re-implementation of max
int max2(int x, int y) where {x ≤ ret, y ≤ ret} { return maxMod(x, y, new Id()); }
```

Class `Id` implements an identity modification. Its `modify` method inherits the constraints of the `Modifier` interface. In LJGS, freshly create objects are always public. By passing a new `Id` instance to `maxMod`, `max2`'s signature is as flexible as that of `max`: the constraint `ymod` $\leq$`ret` of `maxMod` is trivially satisfied and can be omitted from the signature of `max2`.

Semantically, the result of `max2`, or of `max` for that matter, always has a security level that is the least upper bound of `x` and `y`. Thus, calling `max2` with statically or dynamically typed arguments makes little difference. In the following call to `maxMod`, however, a static call is more conservative than a dynamic one.

```
class Erase extends Modifier {int[•] def; int modify(int y) { return def; }}
/* ... */ z = maxMod(42, y, new Erase(0)); /* ... */
```

Here we use an `Erase` modifier that replaces the value of `y` value with a public constant. The signatures of `maxMod` and `Erase.modify` still assert a dependency on the argument `y` but dynamically the connection is cut by the particular implementation of `Erase.modify`. Thus,

$$
\begin{array}{lll}
prog & ::= & cld_1 \,..\, cld_n s \\
cld & ::= & \textbf{class}\, C\, \textbf{extends}\, cl\{F_1[a_1] \,..\, F_n[a_n]\, md_1 \,..\, md_m\} \quad cl ::= C \mid \textbf{Object} \\
a & ::= & \bullet \mid A \mid \star \quad md ::= M(var_1, \,..\, , var_n)\, \textbf{where}\, \mathcal{S}\, \textbf{and}\, \mathcal{G}\{s\, \textbf{return}\, y\} \\
x, y & ::= & var \mid \textbf{this} \\
e & ::= & x \mid x.F \mid N \mid x + y \mid \textbf{new}\, C(x_1, \,..\, , x_n) \mid (a \Leftarrow a')x \\
s & ::= & var = e \mid x.F = y \mid var = x.M(y_1, \,..\, , y_n) \mid s;\ s \\
 & & \mid\ \textbf{if}\, [sl, \mathcal{L}, \mathcal{G}](x == y)\{s\}\{s\} \mid \textbf{while}\, [sl, \mathcal{L}, \mathcal{G}](x == y)\{s\} \mid (a \Rrightarrow a')\{s\} \\
sl & ::= & \text{unique identifiers for branching statements}
\end{array}
$$

Figure 5 Syntax of LJGS.

the following call succeeds without security problems when `printPublic` is a low-security sink and `secret` a static high-security variable.

```
y = (⋆ ⇐ HIGH) secret; z = maxMod(42, y, new Erase(0)); printPublic((LOW ⇐ ⋆)z);
```

A corresponding static call without the typecasts would be rejected by the type checker: `maxMod`'s signature would type `z` as `HIGH`, even though no information flows from `y` to `z`.

3 Syntax of LJGS Programs

LJGS extends Lightweight Java with annotations for security types and casts.

To better focus on the security aspects and to avoid notational clutter, we omit the Java types and signatures for fields, local variables, and methods; we only write the annotations relevant for security typing.[4] Nevertheless, we assume that all programs are well-typed Java programs where types, signatures, and declarations of local variables are erased.

An LJGS program (see Figure 5) consists of a set of class definitions, cld, followed by a statement s, the entry point to the program. Class definitions consist of field declarations, $F_i[a_i]$, and method declarations, md_i. Classes form a hierarchy with **Object** at its root. For simplicity, all method and field names are unique and we assume that the relationship of method overrides is externally defined.

A field declaration relates a field name with a security type a. A security type is either the *public type* $\bullet$, a static security level A, or the *dynamic type* $\star$.

A method definition declares the method name, M, a list of parameters, a set of *method constraints* $\mathcal{S}$, the global effect $\mathcal{G}$, a statement s that serves as method body, and a single local variable y for the return value. We write $\text{params}\,(M)$, $\text{constraints}\,(M)$, and $\text{effects}\,(M)$ to refer to the parameters, method constraints and effects, respectively. Method constraints and effects are discussed in detail in Sections 4.1 and 4.3. Further local variables are implicitly defined by their first assignment.

A variable x is either user-defined, var, or references the receiver of the method call, **this**. An expression e can be a variable access, x, field access $x.F$, an integer constant, N, integer addition, $x + y$, object instantiation, $\textbf{new}\, C(x_1, \,..\, , x_n)$, or a value cast. A cast expression $(a \Leftarrow a')x$ converts the value of x from *source type* a' to *destination type* a. We omit a **null** value for simplicity. It would be straightforward to add, however, as **null** behaves like an integer constant with respect to information flow.

[4] In an implementation, we have to write Java types and signatures as well as security annotations.

$$
\begin{array}{lcl}
\mathcal{S} & ::= & \{sc_1, .., sc_n\} \\
sc & ::= & st \leq st \mid st \sim st \\
st & ::= & a \mid x \mid \textbf{ret}
\end{array}
\qquad
\begin{array}{lcl}
\mathcal{C} & ::= & \{c_1, .., c_n\} \\
c & ::= & sec \leq sec \mid sec \sim sec \\
sec & ::= & st \mid \alpha \\
tvar & ::= & x \mid \textbf{ret} \mid \alpha
\end{array}
$$

Figure 6 Components of method constraints and typing constraints.

A *local update*, $var = e$ assigns the value of an expression e to a variable. A *field update* $x.F = y$ writes the value of y to field F of the object referenced by x. A *method call* $var = x.M(y_1, .., y_n)$ stores the result of calling method M with arguments $y_1, \ldots, y_n$ in variable var. The **if** and **while** statements are standard except for three annotations, a *source location* sl and *effects* $\mathcal{L}$ and $\mathcal{G}$. The role of the effects is further explained in Sections 4.3 and 5.4. The external write effect analysis that backs the dynamic IFC uses source locations to identify branching statements. The role of the effect analysis is further explained in Section 5.3. The examples of Section 2 do not show locations or effects on **if** statements as they can be inferred. Finally, the *context cast* statement $(a \Rrightarrow a')\{s\}$ embeds a computation with *inner* program counter type a' into a context with *outer* program counter type a.

4 Security Constraints and Typing Rules

In LJGS, a programmer specifies information-flow properties by providing suitable method signatures. As illustrated in Section 2, a signature relates parameters, return values, and side effects of methods with security types. The type system ensures that the signature specifications are adequate and that operations depending on statically typed parameters and fields have no security leaks. Its design also ensures that the security levels of statically typed values need not be tracked at run-time.

Security types form a lower semi-lattice induced by the following ordering.

▶ **Definition 1** (Partial order on security types).

$$\bullet \leq a \qquad A \leq A' \text{ if } A \sqsubseteq A' \qquad \star \leq \star$$

By embedding the lattice of security levels into security types, LJGS supports the usual notion of security subtyping: values with a low security level are implicitly promoted to a higher one and contexts with a low-security program counter type admit computations that perform side effects on a higher security level. To ensure a clean boundary between static and dynamic code, implicit conversion between static types and the dynamic type is not allowed; that is, the supremum of A and $\star$ is not defined. However, the public security type $\bullet$ may act polymorphically as the dynamic type and the static type at the bottom of the security lattice ($\bot$).

4.1 Method Constraints

The constraints in a method signature represent the information flow dependencies between parameters and return values: flows into the return value are represented by lower bounds on a return symbol, whereas flow restrictions on the arguments are represented by upper bounds on parameter symbols. Figure 6 defines the syntax for method constraints $\mathcal{S}$. A constraint relates its *method constraint types*, st_1 and st_2, either by *subsumption*, $st_1 \leq st_2$, or by *compatibility*, $st_1 \sim st_2$. Method constraint types st are literal security types a or

symbols for a method's formal parameters x (including **this**) and return value, **ret**. For simplicity, we require that all method constraints mention exactly the parameters of their respective methods, as well as **ret** and **this**.

The constraint $st_1 \leq st_2$ ("st_2 subsumes st_1") specifies that st_1 and st_2 should be ordered according to Definition 1. For example, the constraint `HIGH` $\leq$ **ret** requires the return value of a method to be a statically known security level that is at least `HIGH`. The constraint $st_1 \sim st_2$ ("st_1 is compatible with st_2") specifies that if st_1 is dynamic then st_2 should be dynamic or public and vice versa. If both components are static, then they are compatible.

Compatibility constraints appear rarely in signatures, but here is an example. Consider a method, `const42`, that contains the same statements as `max`, but subsequently cuts the flow from parameters to the result by overwriting the `x` with a constant.

```
int const42(int x, int y) where { x ∼ y } {
  if (x ≤ y) {x = y;}
  x = 42; return x;
}
```

Although `const42` ignores the parameters and does not constrain `ret`, the parameters `x` and `y` cannot have arbitrary types: If `const42` is called with a dynamic first argument, the second argument needs to be dynamic or public to enable the run-time system to track implicit flows arising from the conditional. The compatibility constraint $x \sim y$ expresses this requirement.

4.2 Typing Constraints and Constraint Interpretation

To type-check a method's body against its signature, LJGS' typing rules (cf. Section 4.4) generate sets of *typing constraints*, $\mathcal{C}$. Typing constraints c (see Figure 6) are like method constraints but relate *statement constraint types*, *sec*, which extend method constraint types with *local type variables* α. Local type variables represent the flow-sensitive security types for the local variables before and after the statements of a method body. A method body complies to a signature if the generated typing constraints refine the information flow between parameters, fields and return value that the signature represents.

We now formalize the intuitive notion of constraints that we used in the examples in Section 2 and Section 4.1. In the following, we let $\mathcal{C}$ range over sets of typing constraints, α, β, γ over local type variables, and we write $\alpha^\dagger$ for a fresh local type variable. We also let *tvar* range over statement constraint types that are not literal security types (cf Figure 6).

▶ **Definition 2** (Assignment, solution, solvability)**.** An assignment is a total function from type variables to security levels. An assignment θ is extended to a total function from components to security levels, θ^*, by mapping security levels to themselves:

$$\theta^*(sec) = a \quad \text{if } sec = a \qquad\qquad \theta(tvar) \quad \text{if } sec = tvar$$

An assignment θ is a *solution* of a constraint set $\mathcal{C}$, written $\theta \models \mathcal{C}$, iff (i) for all constraints $sec_1 \leq sec_2 \in \mathcal{C}$, it holds that $\theta^*(sec_1) \leq \theta^*(sec_2)$, and (ii) for all constraints $sec_1 \sim sec_2$, it holds that there exists a type a such that $\theta^*(sec_1) \leq a$ and $\theta^*(sec_2) \leq a$.

A constraint of the form $\bullet \leq sec$ imposes no restriction on solutions. A compatibility constraint that contains a static and a dynamic component, like $\star \sim$ `LOW`, is never solvable.

Type checking of LJGS requires that the typing constraints generated for method bodies refine the constraints of the respective signatures. For method- and typing constraints, refinement amounts to checking entailment of constraints modulo local type variables.

▶ **Definition 3** (Refinement of typing constraints)**.** Let $\mathcal{C}_1, \mathcal{C}_2$ be two constraint sets. $\mathcal{C}_2$ *refines* $\mathcal{C}_1$, written $\mathcal{C}_2 <: \mathcal{C}_1$, if for each solution $\theta \models \mathcal{C}_1$ there exists an assignment θ' such that (i) $\theta' \models \mathcal{C}_2$, (ii) $\theta'(x) = \theta(x)$ for all parameters x, and (iii) $\theta'(\mathbf{ret}) = \theta(\mathbf{ret})$.

Example: the constraints $\{\texttt{LOW} \leq \mathbf{ret}\}$ and $\{\texttt{LOW} \leq \alpha, \alpha \leq \mathbf{ret}, \texttt{HIGH} \leq \beta\}$ both refine the (more restrictive) constraint $\{\texttt{HIGH} \leq \mathbf{ret}\}$.

4.3 Effects

Typing of LJGS method bodies includes the generation of two kinds of effects, $\mathcal{L}$ and $\mathcal{G}$. The *local effect* $\mathcal{L}$ is a set of variables. It appears as annotation on branching statements. If a local variable is listed in $\mathcal{L}$ then the branching statement may modify that variable and thus potentially taint it with information from the branch condition. The run-time system uses $\mathcal{L}$ to update local variables in untaken branches.

The *global effect* $\mathcal{G}$ is a set of security types. It appears on method definitions and branching statements. If a security type is listed in $\mathcal{G}$, then the method or branching statement may perform a side effect that leaks information into a (globally) accessible field of that type. The type system uses $\mathcal{G}$ to track implicit flows to the heap.

There is a refinement relation for global effects that amounts to checking subsumption contravariantly on the contained security types.

▶ **Definition 4** (Subsumption of global effects)**.** $\mathcal{G}_2$ *refines* $\mathcal{G}_1$, written $\mathcal{G}_2 <: \mathcal{G}_1$, if for all $a \in \mathcal{G}_2$ there exists $a' \in \mathcal{G}_1$ such that $a' \leq a$.

For example, the global effects $\{\mathit{high}, \star\}$ and $\{\mathit{low}, \mathit{high}, \star\}$ both refine $\{\mathit{low}, \star\}$. In contrast $\{\mathit{high}, \star\}$ does note refine $\mathcal{G} = \{\mathit{low}\}$, as there is no type in $\mathcal{G}$ that is smaller or equal to $\star$.

4.4 Typing Rules

We are now in a position to define the rules for LJGS. In addition to the typing requirements for Java-like languages, well typed LJGS programs satisfy three conditions: all method constraints are satisfiable, the signatures of overriding methods of every subclass refine the corresponding signatures of its superclass, and, for every method, LJGS' statement typing rules generate constraints and effects that refine the method's signature.

The following definition captures the refinement requirement for subclassing:

▶ **Definition 5** (Well-typed class hierarchy)**.** Let constraints (M) and effects (M) be the constraints and the effects that are given by the signature of method M. Method M_1 *refines* method M_2 if constraints $(M_1) <:$ constraints (M_2) and effects $(M_1) <:$ effects (M_2).

The class hierarchy of an LJGS program is well-typed if for every method M_1 that overrides a method M_2 method M_1 also refines method M_2.

The following judgment summarizes the requirements for method bodies:

$$\frac{\begin{array}{c}\Gamma_1 = [\mathit{var}_1 \mapsto \mathit{var}_1, .., \mathit{var}_n \mapsto \mathit{var}_n, \mathbf{this} \mapsto \mathbf{this}] \\ \gamma \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, (\mathcal{L}, \mathcal{G}') \qquad \alpha = \Gamma_2(y) \qquad \mathcal{C} \cup \{\alpha \leq \mathbf{ret}\} <: \mathcal{S} \qquad \mathcal{G}' <: \mathcal{G}\end{array}}{\vdash M(\mathit{var}_1, .., \mathit{var}_n)\ \mathbf{where}\ \mathcal{S}\ \mathbf{and}\ \mathcal{G}\{s\ \mathbf{return}\ y\}}$$

Its central requirement is the judgment $\gamma \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}$ which generates constraints $\mathcal{C}$ and effects $\mathcal{E}$ for well-typed statements. Effects $\mathcal{E}$ have the form $(\mathcal{L}, \mathcal{G})$. Γ_1 and Γ_2 are typing environments that map local variables to the type variables. The environments connect a local variable with the variable that represents its type before and after a particular

statement. Together with $\mathcal{C}$, the *initial environment* Γ_1 describes the typing constraints for local variables before s is executed, whereas the *final environment* Γ_2 describes local variable types after executing s. The type variable γ represents the type of the program counter. The method typing judgment pre-initializes the environment for the parameters and checks if the generated constraints and effects refine the constraints and effects of the signature.

4.4.1 Statement Typing: Overview

Section 4.4.2 explains some of the typing rules in detail. A complete listing of rules can be found in a technical report[5]. Although the details of the typing rules may seem complex the basic principles behind the constraint generation are rather simple: To track explicit flows in assignment statements, the rules generate constraints where each read variable or field is a lower bound to the variable written. For example, the statement `x = z.F; x = x + y` yields constraints that include $\mathcal{C} = \{z \leq \beta_1, a \leq \beta_1, \beta_1 \leq \beta_2, y \leq \beta_2\}$. Here, a is the type of the field F, z and y are parameters and β_1, β_2 are the type variables that the environment maps to x after the first and second assignment, respectively. The typing rules consider implicit flows by checking with the program counter variable γ. Each assignment generates an additional constraint where the program counter variable is a lower bound of the updated variable. In the example, $\mathcal{C}$ would be extended with the constraints $\gamma \leq \beta_1, \gamma \leq \beta_2$. If, for example, the result of a branch condition flows into the branches of an **if** statement, a new program counter variable is introduced. Program counter variables establish the implicit flows from local variables and parameters to results and field writes. For example, the statement **if** `(x = y) { z.F = 42; }`, where `x, y, z` are parameters and the type of field F is a, and the initial program counter type is γ, generates the constraints $\mathcal{C}_2 = \{\gamma \leq \gamma', x \leq \gamma', y \leq \gamma', \gamma' \leq z, \gamma' \leq a\}$. A method constraint like $\{x \leq z, y \leq z, z \leq a\}$ that subsumes $\mathcal{C}_2$, take the flows through γ' into account without mentioning the program counter variables.

Casts hide information flows from the type system. The constraints generated by assignments with value casts do not connect the type of the right-hand-side variable with that of the result. Instead, they connect the right-hand-side type with the source type of the cast and the cast's destination type with the result. For example, the statement `x = (`$\star \Leftarrow$ `LOW)y` has the constraints $\{\gamma \leq \alpha_1, y \leq \texttt{LOW}, \star \leq \alpha_2\}$ which do not relate the type of `y` with that of `x` (α_1). Similarly to value casts, context casts break up the connection between program counters and insert their respective source and destination types, instead.

4.4.2 Statement Typing: Details

Figure 7 gives the rules for assignments. The rule for local assignment, ST-LOCAL, yields the constraints for a fresh type variable, $\alpha^\dagger$, that represents the type of the updated local variable after the assignment. The final type environment is updated accordingly. An expression typing judgment generates the constraints for explicit flows to $\alpha^\dagger$ while additional constraint $\gamma \leq \alpha^\dagger$ takes care of implicit flows. Writing to variable *var* generates a singleton local effect $\{var\}$. No global effects are generated, as local updates are not visible outside of the method.

Rule ST-PUTFIELD types write operations to a field F. It requires that F's type, indicated by $\text{fsec}(F)$, subsumes the type of the source variable, the type of the accessed object reference x, as well as the type of the program counter. The statement's effect is a global effect with F's type. As no local variables are modified, initial and final environments are identical.

[5] `http://proglang.informatik.uni-freiburg.de/projects/gradual/ljgs/ecoop2016-tr.pdf`

$\boxed{\gamma \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}}$

ST-LOCAL
$$\frac{\Gamma_1, \alpha^\dagger \vdash e : \mathcal{C} \qquad \Gamma_2 = \Gamma_1[var \mapsto \alpha^\dagger]}{\gamma \vdash var = e : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C} \cup \{\gamma \leq \alpha^\dagger\}, (\{var\}, \varnothing)}$$

ST-PUTFIELD
$$\frac{\mathcal{C} = \{\Gamma(x) \leq \mathrm{fsec}\,(F), \Gamma(y) \leq \mathrm{fsec}\,(F), \gamma \leq \mathrm{fsec}\,(F)\}}{\gamma \vdash x.F = y : \Gamma \Rightarrow \Gamma, \mathcal{C}, (\varnothing, \{\mathrm{fsec}\,(F)\})}$$

Figure 7 Statement typing: updates.

RT-NEW
$$\frac{a_1 \,..\, a_n = \mathrm{fieldsecs}\,(C) \qquad \mathcal{C} = \{\Gamma_1(x_1) \leq a_1, \,..\, , \Gamma_1(x_n) \leq a_n\}}{\Gamma, \alpha \vdash \mathbf{new}\, C(x_1, \,..\, , x_n) : \mathcal{C}}$$

RT-LOCAL
$$\frac{\mathcal{C} = \{\Gamma(x) \leq \alpha\}}{\Gamma, \alpha \vdash x : \mathcal{C}}$$

RT-CAST
$$\frac{a \lesssim a' \qquad \mathcal{C} = \{\Gamma(x) \leq a, a' \leq \alpha\}}{\Gamma, \alpha \vdash (a' \Leftarrow a)x : \mathcal{C}}$$

$$\frac{}{\star \lesssim a} \qquad \frac{}{a \lesssim \star} \qquad \frac{}{\bullet \lesssim A} \qquad \frac{}{\bot \lesssim \bullet}$$

Figure 8 Expression typing $\boxed{\Gamma, \alpha \vdash e : \mathcal{C}}$ and castability $\boxed{a \lesssim a'}$.

Figure 8 gives some of the constraint generation rules for expressions. The constraints for object allocation, rule RT-NEW, state that constructor arguments flow into their corresponding field. The meta-function **fieldsecs** yields the security types of a class' fields. The new object reference is considered public on creation. Constants (rule not shown) are also considered public in LJGS and impose no constraints. For variable access, rule RT-LOCAL requires that the result type subsumes the type of the accessed local variable. The rules for addition and field access (not shown) are similar. The rule for casts, RT-CAST, requires that the result type subsumes the destination type of the cast and that the source type subsumes that of the accessed variable. Additionally the source type needs to be *castable* into the destination type, written $a \lesssim a'$. Castability, defined in Figure 8, rules out casts that are either unnecessary or are guaranteed to fail: casts from A to A' trivially succeed when $A \sqsubseteq A'$ and fail when $A \not\sqsubseteq A'$. Casts from A to $\bullet$ also trivially fail when $A \neq \bot$.

Figure 9 shows rules for statements that create new contexts. Rule ST-CALL in Figure 9 covers method calls. It extracts the formal parameters, the method constraints, and the effects of the callee M with the operations $\mathrm{params}\,(M)$, $\mathrm{constraints}\,(M)$, and $\mathrm{effects}\,(M)$. Then, it instantiates the information flows stated in the method constraints for the current context by simultaneous substitution of parameters with arguments and of **ret** with the fresh local type variable $\alpha^\dagger$. Otherwise the rules behave like assignments. The global effect of the method signature is treated analogously as in rule ST-PUTFIELD. The operation $\prod_\gamma \mathcal{G} := \{\gamma \leq a \mid a \in \mathcal{G}\}$ generates the corresponding constraints. The rule for conditionals, ST-IF, types the branches s_1 and s_2 under a fresh program counter variable $\beta^\dagger$. Types assigned to $\beta^\dagger$ need to subsume the old program counter type γ and the types of the condition variables x and y (cf. $\mathcal{C}'$). The effects of the conditional is the union of the effects of both branches. The final environment is a join of the final environments of s_1 and s_2. The join operation $\Gamma_2' \sqcup \Gamma_2''$ generates additional constraints to be included in the final constraints $\mathcal{C}$.

$$\textsc{st-call}\quad \frac{\mathcal{S} = \text{constraints}\,(M) \qquad \mathcal{G} = \text{effects}\,(M) \qquad var_1, \ldots, var_n = \text{params}\,(M) \\ \mathcal{C}' = \mathcal{S}[\mathbf{this} \mapsto \Gamma_1(x), var_1 \mapsto \Gamma_1(x_1), \ldots, var_n \mapsto \Gamma_1(x_n), \mathbf{ret} \mapsto \alpha^\dagger] \\ \mathcal{C} = \mathcal{C}' \cup \{\Gamma_1(x) \leq \alpha^\dagger, \gamma \leq \alpha^\dagger\} \cup \bigsqcap_\gamma \mathcal{G} \qquad \Gamma_2 = \Gamma_1[var \mapsto \alpha^\dagger]}{\gamma \vdash var = x.M(x_1, \ldots, x_n) : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, (\{var\}, \mathcal{G})}$$

$$\textsc{st-if}\quad \frac{\beta^\dagger \vdash s_1 : \Gamma_1 \Rightarrow \Gamma_2', \mathcal{C}_1, (\mathcal{L}_1, \mathcal{G}_1) \\ \beta^\dagger \vdash s_2 : \Gamma_1 \Rightarrow \Gamma_2'', \mathcal{C}_2, (\mathcal{L}_2, \mathcal{G}_2) \qquad \mathcal{C}' = \{\gamma \leq \beta^\dagger, \Gamma_1(x) \leq \beta^\dagger, \Gamma_1(y) \leq \beta^\dagger\} \\ \Gamma_2, \mathcal{C}'' = \Gamma_2' \sqcup \Gamma_2'' \qquad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}' \cup \mathcal{C}'' \qquad \mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2 \qquad \mathcal{G} = \mathcal{G}_1 \cup \mathcal{G}_2}{\gamma \vdash \mathbf{if}\ [sl, \mathcal{L}, \mathcal{G}](x == y)\{s_1\}\{s_2\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, (\mathcal{L}, \mathcal{G})}$$

$$\textsc{st-cxcast}\quad \frac{\beta^\dagger \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}', (\mathcal{L}, \mathcal{G}) \qquad \mathcal{C} = \{a \leq \beta^\dagger, \gamma \leq a'\} \cup \mathcal{C}' \qquad a' \lesssim a}{\gamma \vdash (a' \Rrightarrow a)\{s\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, (\mathcal{L}, \{a'\})}$$

Figure 9 Statement typing: branching, method calls and context casts, $\boxed{\gamma \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}}$.

▶ **Definition 6** (Join of typing environments)**.** An environment Γ together with a constraint set $\mathcal{C}$ form the join of two environments Γ_1 and Γ_2, written $\Gamma, \mathcal{C} = \Gamma_1 \sqcup \Gamma_2$, iff

$$\mathcal{C} = \{\Gamma_1(x) \leq \alpha_x \mid x \in \text{dom}(\Gamma_1)\} \cup \{\Gamma_2(x) \leq \alpha_x \mid x \in \text{dom}(\Gamma_2)\}$$
$$\Gamma = \{x \mapsto \alpha_x \mid x \in \text{dom}(\Gamma_1) \cup \text{dom}(\Gamma_2)\}$$

where $\{\alpha_x \mid x \in \text{dom}(\Gamma_1) \cup \text{dom}(\Gamma_2)\}$ is a set of fresh type variables, one for each variable in the domain of Γ_1 and Γ_2.

Rule ST-WHILE (not shown) works similarly to rule ST-IF. Rule ST-CXCAST deals with the cast between program counters: The program counter type of the cast context needs to subsume the outer type a' given in the cast. The outer type also defines the global effect of the statement. The body of the cast is typed under a fresh program counter type variable $\beta^\dagger$. The constraints require that $\beta^\dagger$ subsumes the inner type a. Together with the typing rules ST-PUTFIELD and ST-CALL, this restriction guarantees that the global effect $\mathcal{G}$ of the body is subsumed by $\{a\}$. Also, a' has to be castable to a.

The rule for sequences (not shown) is unsurprising: it combines constraints and effects of its components, as expected.

5 Dynamics

To enforce non-interference in dynamically checked code, LJGS' dynamics propagate and check run-time security labels. Before explaining the full operational semantics, we first discuss the interpretation of security labels and their operations.

5.1 Security Labels

Security labels (see Figure 10) are the mirror image of security types: a security label is either the static label, S, a dynamic label, D(A), carrying a security level or, the *public label*, •. The level of a static label is absent at run-time because it has been checked by the type

$$
\begin{array}{rcl@{\qquad}rcl}
\varsigma, \ell, g & ::= & \mathrm{S} \mid \mathrm{D}(A) \mid \bullet & v & ::= & rv[\varsigma] \\
s & ::= & \ldots \mid \mathbf{done} & rv & ::= & \mathit{oref} \mid N \\
E & ::= & \langle\rangle \mid E;\ s \mid \mathbf{cx}\,[\mathcal{L}, \mathcal{G}, \mathcal{R}, \ell]\{E\} & L & ::= & \varnothing \mid L \oplus x \mapsto v \\
 & \mid & \mathbf{cx}\,[a/\ell/g \Rrightarrow a'/\ell/g']\{E\} & & \mid & L \oplus x \mapsto \mathbf{null} \\
 & \mid & \mathbf{call}\,[M, var, x, y_1, \ldots, y_n, L]\{E\} & \mu & ::= & \varnothing \mid \mu \oplus \mathit{oref} \mapsto \mathit{obj} \\
rs & ::= & E(s) & & &
\end{array}
$$

Figure 10 Dynamic domains and run-time statements.

system already. The dynamic subset of security labels form a lattice based on the attached security levels. The entire set of labels forms a lower semi-lattice with $\bullet$ as bottom element, analogously to the semi-lattice of security types.

During the execution of an LJGS program, every value carries a *value label* ς. To track implicit flows, the semantics maintains two labels for each execution context, a *local program counter label*, ℓ, and a *global program counter label*, g. For value labels, this security level indicates the current run-time confidentiality of a labeled value. A program counter label approximates the confidentiality of information that implicitly flows into the current program counter. Thus, the dynamic label on a program counter gives a lower bound on the side effects.

When information flows into a value or program counter from multiple sources, the dynamic semantics employs a partial join operation on the labels involved. The judgment $\varsigma := \varsigma_1 \sqcup \varsigma_2$ determines the label ς that joins ς_1 and ς_2.

$$
\frac{}{\mathrm{S} := \mathrm{S} \sqcup \mathrm{S}} \qquad \frac{}{\mathrm{D}(A_1 \sqcup A_2) := \mathrm{D}(A_1) \sqcup \mathrm{D}(A_2)} \qquad \frac{}{\varsigma := \bullet \sqcup \varsigma} \qquad \frac{}{\varsigma := \varsigma \sqcup \bullet}
$$

Joining static labels is trivial. Dynamic labels are joined by joining the security levels they contain. The public label is a neutral element for the join operation. Joining static and dynamic labels is undefined, as the security level of a static labeled entity cannot be recovered at run-time. A well typed LJGS program only performs well-defined joins.

5.2 Configurations

The execution of a method manipulates *configurations* $rs/L/\mu$ of run-time statements rs, *stack frames* L, and *heaps* μ. Run-time statements, defined in Figure 10, consist of execution contexts E applied to source statements s. For technical reasons, the statements previously defined in Section 3 need to be extended with the marker statement **done**. An execution context is a hole $\langle\rangle$, a sequence of an execution context and a source statement, a *run-time security context* $\mathbf{cx}\,[\mathcal{L}, \mathcal{G}, \mathcal{R}, \ell]\{E\}$, a *cast context* $\mathbf{cx}\,[a/\ell/g \Rrightarrow a'/\ell'/g']\{E\}$, or a *calling context* $\mathbf{call}\,[M, var, x, x_1, \ldots, x_n, L]\{E\}$. The run-time statement $\langle\mathbf{done}\rangle$ signals that execution of the current context is complete. A sequence context focuses the first component of a statement sequence. A run-time security context remembers metadata that is needed to track the dynamic implicit flows that arise from branching statements and virtual method calls: the program counter label ℓ that is active during its execution, and local and global effects, $\mathcal{L}$, $\mathcal{G}$, that were analyzed for its body. It also remembers a set of *field references* $\mathcal{R}$ which we explain in Sections 5.3 and 5.4. A cast context stores the old and new program counter types and labels during the execution of a block subject to a context cast. A calling context stores the method name and stack frame of the callee and the return variable of the caller during a method call. It also contains the method call receiver x and the call arguments $x_1, \ldots, x_n$ for technical reasons.

A stack frame is a finite map from local variables to *values*. Stack frames have a fixed size and bind exactly the local variables that occur in the body of their method. Uninitialized variables are bound to **null**.

A value consists of a *raw value* rv and value label ς. A raw value is either a *number* $N \in \mathbb{N}$ or an *object reference*, *oref*. A heap is a finite map from *references* to *objects*. An object consists of its run-time class C and the values of its fields.

5.3 Effect Analysis Support for Dynamic IFC

Information flow control in the dynamic fragment of LJGS copes with implicit flows by relying on statically analyzed information about write effects in untaken control flow branches.[6] As in Russo and Sabelfeld's work [23] the basic principle is to upgrade the labels of dynamic variables and fields if they may be updated in the untaken branches.

For upgrades of local variables the typing rules generate sufficient information with the local effect $\mathcal{L}$. For object oriented languages like LJGS, obtaining precise information about heap write effects is more involved. Including such an analysis in LJGS would go beyond the scope of this paper. Instead, we rely on an external static analysis for write effects that provides (abstract) references to potentially updated fields in branching statements and method calls. In the semantics, we encapsulate the analysis result in the meta-function call $\mathcal{R} = \text{updaterefs}\,(sl, L, \mu)$. The function updaterefs takes as arguments the location of a branching statement (or the name of a method), a stack frame, and a heap, and returns a set $\mathcal{R}$ of (concrete) field references $oref.F$. The result $\mathcal{R}$ contains references to all fields that may be updated by the statement labelled sl, the concretization of the externally analyzed write effect for L and μ. The technical report gives a precise specification of updaterefs.

There are a number of interprocedural points-to analyses for languages with heap references [18, 28] from which a write effect analysis suitable for implementing updaterefs can be derived. We limit the static analysis to write effects for simplicity; it should also be possible to directly incorporate points-to information into the dynamic IFC policy, as illustrated by Moore and Chong [20], to gain precision.

5.4 Reduction rules

The judgment $\ell;\ g \vdash rs/L/\,\mu \longrightarrow rs'/L'/\,\mu'$ defines the small step reduction of a configuration under the local program counter label ℓ and global program counter label g. Apart from the rules for casts, reduction mostly follows the principles of other dynamic IFC systems [3, 23]. Figures 11 and 12 show some of the rules. The technical report contains the full semantics.

Figure 11 shows the rules for evaluating expressions and for updating local variables. The judgment $\vdash e/L/\,\mu \Downarrow v/\,\mu'$ evaluates expression e to a result and an updated heap. Rule STEP-UPD-PLUS illustrates evaluation. It reads the operands from the stack frame L, and returns a raw value, here $N + N'$, that has an updated security label attached. The updated label is the join of the operands' labels. Allocations, covered by rule STEPUPD-NEW, extend the heap with a fresh reference which has the public label. Other expressions that are not shown work similarly. A cast expression converts the value label of its subject according to the label conversion judgment $a \Rrightarrow a' \vdash \varsigma \Rrightarrow \varsigma'$, also defined in Figure 11. Label conversion

[6] IFC techniques where the label propagation relies on static analysis are typically called *hybrid IFC*. Given our setting, we still refer to these approaches as dynamic IFC because they are based on run-time propagation of security labels and because LJGS employs such an approach in the dynamically typed fragments of programs.

$$\textsc{stepupd-plus}\ \frac{N[\varsigma_1] = L[x] \quad N'[\varsigma_2] = L[y] \quad \varsigma := \varsigma_1 \sqcup \varsigma_2}{\vdash x + y/L/\,\mu \Downarrow N + N'[\varsigma]/\,\mu}$$

$$\textsc{stepupd-new}\ \frac{v_1 = L[x_1] \;..\; v_n = L[x_n] \quad \mathit{oref} = \mathit{fresh} \quad v' = \mathit{oref}[\bullet] \quad \mu' = \mu \oplus \mathit{oref} \mapsto \{C, v_1 .. v_n\}}{\vdash \mathbf{new}\, C(x_1, .., x_n)/L/\,\mu \Downarrow v'/\,\mu'}$$

$$\textsc{stepupd-cast}\ \frac{rv[\varsigma] = L[x] \quad a' \Rrightarrow a \vdash \varsigma \Rrightarrow \varsigma'}{\vdash (a \Leftarrow a')x/L/\,\mu \Downarrow rv[\varsigma']/\,\mu}$$

$$\textsc{estep-local}\ \frac{\vdash e/L/\,\mu \Downarrow rv[\varsigma]/\,\mu' \quad \varsigma' := \varsigma \sqcup \ell \quad L' = L[var \mapsto rv[\varsigma']]}{\ell;\ g \vdash \langle var = e\rangle/L/\,\mu \longrightarrow \langle \mathbf{done}\rangle/L'/\,\mu'}$$

$$\frac{}{a \Rrightarrow a \vdash \varsigma \Rrightarrow \varsigma} \qquad \frac{}{A \Rrightarrow \star \vdash \varsigma \Rrightarrow \mathrm{D}(A)} \qquad \frac{A \sqsubseteq A'}{\star \Rrightarrow A' \vdash \mathrm{D}(A) \Rrightarrow \mathrm{S}} \qquad \frac{}{\bullet \Rrightarrow A \vdash \varsigma \Rrightarrow \mathrm{S}}$$

$$\frac{}{\bullet \Rrightarrow \star \vdash \varsigma \Rrightarrow \mathrm{D}(\bot)} \qquad \frac{}{\bot \Rrightarrow \bullet \vdash \mathrm{S} \Rrightarrow \bullet} \qquad \frac{}{\star \Rrightarrow \bullet \vdash \mathrm{D}(\bot) \Rrightarrow \bullet}$$

Figure 11 Local updates (excerpt) $\boxed{\vdash e/L/\,\mu \Downarrow v/\,\mu'}$ and label conversion $\boxed{a' \Rrightarrow a \vdash \varsigma' \Rrightarrow \varsigma}$.

changes a label ς that corresponds to a cast's source type a to a label ς' corresponding to the destination type a'. A trivial cast results in a trivial label conversion. The conversion from a static type to the dynamic type creates a dynamic label that contains the source type as security level. A static-to-dynamic conversion always succeeds in a well-typed LJGS program. A conversion from dynamic to a static type A' returns the static label, if the dynamic source label contains a security level subsumed by A. Otherwise, the static-to-dynamic conversion fails. The public type may be converted to a dynamic type and a low-security label or any static type. Converting to the public type requires either a static source type of bottom or a dynamic bottom label. Finally, rule ESTEP-LOCAL stores the result of an evaluation in a local variable after joining its label with the current program counter label.

Figure 12 shows illustrative cases of reductions that enter and leave contexts. Rule ESTEP-IF-TRUE covers if-statements where the branch condition is satisfied. Selecting a branch potentially creates an implicit flow from the condition to the execution context of the branch. Thus, the reduction results in a run-time security context that stores a program counter label ℓ'' which includes the value labels of the operands and the label of the program counter in the outer context. The run-time context also stores the effect annotations of the conditional and the field reference set $\mathcal{R}$ that points to the heap locations that are potentially modified during the execution the untaken branch s_2. The rules for failing conditions, while loops and method calls (not shown) are similar. Method calls additionally create a calling context $\mathbf{call}\,[M, var, x, y_1, .., y_n, L]\{E\}$ that initializes new stack frames and copies the method result into the caller's stack frame in a straightforward way.

Rule ESTEP-CX-DONE exits the run-time context and, to counter illegal implicit flows, performs a preemptive upgrade of local variables and fields mentioned in the context. The meta functions $\mathbf{upgr}(\mu, \mathcal{R}, \mathcal{G}, \ell)$ and $\mathbf{upgr}(L, \mathcal{L}, \mathcal{G}, \ell)$ perform these upgrades for all field references in $\mathcal{R}$ and all (dynamic) variables in $\mathcal{L}$, respectively. The functions use the program counter label for upgrading, if it is dynamic. But, due to context casts, upgrades may also

ESTEP-IF-TRUE

$$\frac{rv[\varsigma_1] = L[x] \qquad rv[\varsigma_2] = L[y] \qquad \varsigma' := \varsigma_1 \sqcup \varsigma_2 \qquad \ell'' := \ell \sqcup \varsigma' \qquad \mathcal{R} = \text{updaterefs}\,(sl, L, \mu)}{\ell;\ g \vdash \langle \textbf{if}\,[sl, \mathcal{L}, \mathcal{G}](x == y)\{s_1\}\{s_2\}\rangle / L / \mu \longrightarrow \textbf{cx}[\mathcal{L}, \mathcal{G}, \mathcal{R}, \ell'']\{\langle s_1\rangle\} / L / \mu}$$

ESTEP-CX-DONE

$$\frac{\mu' = \textbf{upgr}(\mu, \mathcal{R}, \mathcal{G}, \ell') \qquad L' = \textbf{upgr}(L, \mathcal{L}, \mathcal{G}, \ell')}{\ell;\ g \vdash \textbf{cx}[\mathcal{L}, \mathcal{G}, \mathcal{R}, \ell']\{\langle \textbf{done}\rangle\} / L / \mu \longrightarrow \langle \textbf{done}\rangle / L' / \mu'}$$

ESTEP-CXCAST

$$\frac{a \Rrightarrow a' \vdash \ell \Rrightarrow \ell' \qquad a \Rrightarrow a' \vdash g \Rrightarrow g'}{\ell;\ g \vdash \langle (a \Rrightarrow a')\{s\}\rangle / L / \mu \longrightarrow \textbf{cx}[a/\ell/g \Rrightarrow a'/\ell'/g']\{\langle s\rangle\} / L / \mu}$$

Figure 12 Reduction: entering and leaving contexts (excerpt) $\boxed{\ell;\ g \vdash rs / L / \mu \longrightarrow rs' / L' / \mu'}$.

be necessary in static contexts where the program counter label ℓ carries no information. For example, consider the following statement that contains a context cast inside of a conditional:

```
if(this.high) {(HIGH ⇛ ⋆) { this.dyn = 42; }}
```

Here, the type of field `high` is the static level `HIGH` and that of `dyn` is dynamic. According to the IFC principle outlined above, the semantics should upgrade **`this.dyn`** after executing the statement, as a dynamic field is conditionally updated with a high-security program counter. In those situations, the upgrade functions use the greatest lower bound of the global effects $\mathcal{G}$ for upgrading. This choice is safe because the source type of context casts are always reflected in the global effects (cf. typing rule ST-CXCAST). The technical report gives a complete definition of the **upgr** functions.

Context casts, covered by rule ESTEP-CXCAST, determine the local and global program counter labels for executing their body with the same label conversion judgment used for value casts. A cast execution context stores the converted and original types and labels. Reduction under a run-time security context, covered by rule ESTEP-CX-STEP, uses the stored, modified program counter label. The global program counter label is joined with the stored label such that global effects also respect the augmented context. When a run-time security context is completely executed, it is discarded (not shown). Cast contexts (not shown) work similarly, but use the stored converted labels to reduce their bodies. Sequence execution contexts, also not shown, completely execute the first context before moving on to the second.

6 Execution Model

The dynamics of LJGS attach security labels to static and dynamic values alike. Fortunately, LJGS is designed such that static labels can be erased in a realistic implementation. In this section, we sketch a compilation strategy from Java with LJGS security annotations to plain Java. The actual implementation of a compiler to bytecode is ongoing work.

Figure 13 shows an LJGS class with a dynamic field `F`, a statically `LOW` field `G` and a method `m` that includes typecasts. LJGS specific annotations and casts are coloured blue. The Figure also shows, in red, the Java code that an LJGS compiler would produce according to our execution model. The blue parts would be removed from the the compilation result. First, the required datastructures are brought into scope: Line 5 retrieves a `LocalMap locals`

```
1  class C { int F[⋆]; int G[LOW];
2
3   void m(int y) where { y ∼ ⋆ }
4                 and {⋆, LOW} {
5    LocalMap locals = getCall();
6    ObjectMap objects = getObjects();
7    PcStack lPC = new PcStack();
8    PcStack gPC = getGlobalPC();
9    int g = this.G;
10   int x = (⋆ ⇐ HIGH) g;
11   locals.put("x", Levels.HIGH);
12   x = x + y;
13   locals.put(
14     "x", join(locals.get("x"),
15               locals.get("y")));
16   lPC.push(locals.get("x"));
17   gPC.push(locals.get("x"));
18   if [this.F] (x == 5) {
19     this.F = 42;
20     objects.put(this, "F", gPC.get());
21   }
22   objects.put(this, "F",
23    join(objects.get(this, "F"),
24         gPC.get()));
25   lPC.pop(); gPC.pop();
26   int y = this.F
27   locals.put("y", objects.get(this, "F"));
28   cast(locals.get("y"), Level.LOW);
29   this.G = (LOW ⇐ ⋆) y;
30  }
31 }
```

Figure 13 An LJGS class compiled to plain Java code.

that maps local variables to security levels. Each executing method call possesses its own `LocalMap`. The caller initializes the `LocalMap` with the security levels of the dynamic method arguments and publishes it for the callee using the method `setCall` as illustrated with the following code fragment:

```
int x = ... /* some dynamic variable */
LocalMap localsForM = new LocalMap();
localsForM.put("x", locals.get("x"));
setCall(localsForM);
m(x);
```

The globally accessible `ObjectMap objects` (line 6) is a weak map from objects and field names to security levels which stores the dynamic labels of all non-null dynamic fields.

Lines 7 and 8 retrieve the `PcStack` objects to track the levels of dynamic local and global program counters. A `PcStack` is a stack of security levels that stores the level of each dynamic branch condition or method call receiver. The join of all security levels on a stack, calculated by `PcStack.get()`, yields the level of the current, dynamic program counter. The local program counter stack is freshly initialized for the method while the global program counter stack is available through the static method `getGlobalPC()`.

Line 9 is the first statement taken from the original LJGS program. As it is a static update, no instrumentation is needed. Lines 10 and 11 perform the cast from `HIGH` to $\star$. As casts from static to dynamic never fail, the value of `g` is copied to `x`. Line 11 stores the source level mentioned in the cast for `x` in `locals`. Similarly, line 14 updates the local map for `x` with the join of the labels of `x` and `y`, using `locals` and the static method `join`.

Line 18 starts a context with a dynamic program counter. Thus, the dynamic label of the tested variable `x` gets pushed to the local and global `PcStacks`. The field update in line 19 with a low-security constant requires an update to the object map with the label of the global pc (line 20). Line 22 forms the join point for the conditional. To protect against implicit flows, the label of field `F` is joined with the level of the global program counter. Afterwards, line 25 pops the program counter stacks. The cast of the dynamic variable to the static security level `LOW` compiles to lines 28 and 29. The static method `cast` checks if `y`'s label can be converted to `LOW`. If not, as is the case in this example, the program is aborted. Otherwise line 29 performs the field update no further dynamic security tracking of `y`'s value.

The compilation of LJGS can rely on the typing derivation of `m` to determine whether code is dynamic or static and thus whether label tracking code should be emitted: For example, line 9 in Figure 13 does not require tracking code as it is a static update with a public program counter, whereas line 10 is a dynamic update that needs to be tracked.

Polymorphic methods, like method `max` of Figure 1, can be called with dynamic or static arguments. To avoid overhead in the static case, a compilation procedure can generate different versions of `max`, say `max_STATIC_STATIC` and `max_DYN_DYN`, for static and dynamic calls, respectively. The Java method `max_STATIC_STATIC` would not contain any tracking code whereas `max_DYN_DYN` would track all updates and program counters. The corresponding results are not shown here for lack of space but included in the technical report. It is not necessary to generate versions for other combinations of parameters, like `max_STATIC_DYN`, as they are ruled out by the method constraints.

7 Unanalyzable Code

LJGS, as presented in the previous sections, relies on two kinds of static analysis: A type analysis checks the typing rules of Section 4. It requires all classes to be annotated with field types and method signatures, and checks all method bodies for compliance with their signature. Additionally, a write effect analysis supports the dynamic IFC (cf. Section 5.3).

From a practical standpoint, a major use case for gradual typing is the integration of legacy code that is hard to type and analyze statically. In addition, one might expect that dynamically typed code could take advantage of highly dynamic language features like reflection. This section explains how to extend LJGS to accommodate for code that cannot be easily annotated or analyzed. We require, however, that it is possible to *instrument* this unanalyzable code, either by source- or bytecode-transformation or by integrating a corresponding monitor to the virtual machine. We thus do not consider legacy code that is impractical to instrument, like precompiled C-libraries.

7.1 Default, Dynamic Type Annotations

The type-checking of unanalyzeable code can be avoided by assuming dynamic annotations by default and using dynamic IFC during its execution. Consider the following example where `LegacyClass` represents a class that should not be subject to (security-) type-checking.

```
1  class LegacyClass { int legacyField; int legacyMethod(int x, int y) {...}}
2  class LJGSClass {void m() {z = reflectiveCall("any" + "Method", x, y); }}
```

The default annotation for the field `legacyField` would be $\star$ and the default constraints and effect for method `legacyMethod` would be $\{\star \leq \mathbf{ret}, x \leq \star, y \leq \star\}$ and $\{\star\}$, respectively. As long as `legacyMethod` does not use LJGS specific features like casts and only accesses other legacy classes, the method's body will comply with the default signature; type-checking would only generate trivial constraints that classify all entities as dynamic.

Care has to be taken in the presence of abstraction-breaking features like reflection which could allow legacy methods to access statically typed fields and methods of LJGS classes that derive from legacy classes. If these features are required, the run-time enforcement needs to be extended to dynamically block access to non-legacy classes.

A reflective call, as illustrated in line 2, has to be treated in the same way as legacy code. Thus, the run-time enforcement needs to check dynamically if the called method, here `anyMethod`, complies to a default signature.

7.2 Avoiding the Analysis of Write Effects

If a precise write effect analysis is infeasible for legacy code, a more conservative approximation can be used, for example, by assuming that *all* fields of all exposed classes of a legacy Java library are updated when calling a legacy method.

Another possibility is to use a *purely dynamic* enforcement, like Austin and Flanagan's non-sensitive-upgrade policy (NSU, [3]). However, a purely dynamic approach may report false positives that would pose no problems for flow-sensitive static typing [23] whereas a hybrid approach, as the one taken by LJGS, is strictly more flexible.

Instead of falling back to purely dynamic IFC, it is possible to assert a particular write effect for legacy method calls and reflective calls and check compliance dynamically using a dynamic effect analysis, like access permission contracts [15].

8 Correctness

To guarantee that the statics and dynamics presented in the previous sections enforce security according to our attacker model, we need to prove termination insensitive non-interference. As the dynamics only check the security labels of dynamically typed values, non-interference also relies on the soundness of the type system which guarantees that statically typed data is not responsible for program crashes caused by security violations. We always implicitly assume that an LJGS program is well-formed according to the standard typing principles of Java. In particular we rely on the fact that no variables or object fields are accessed uninitialized. Also, we implicitly assume that the class hierarchy in an LJGS program is well-typed, according to Definition 5. Proof sketches of the theorems stated in this section can be found in the technical report.

8.1 Soundness of the type system

Simple typing judgments for values and annotations connect static and dynamic domains.

▶ **Definition 7** (Typing of Dynamic Domains)**.** A **security label** ς **has type** a, written $\vdash \varsigma : a$, if either (i) $\varsigma = \bullet$, (ii) $\varsigma = \mathrm{D}(A)$ and $a = \star$, or (iii) $\varsigma = \mathrm{S}$ and $a = A$. A **security label is typed by a type variable** α under constraints $\mathcal{C}$, written $\mathcal{C} \vdash \varsigma : \alpha$, if there exist a and θ such that $\theta \models (\mathcal{C} \cup \{\alpha \leq a\})$ and $\vdash \varsigma : a$. A **stack frame** L is well-typed under constraints $\mathcal{C}$ and environment Γ, written $\Gamma, \mathcal{C} \vdash L$ if for all bindings $x \mapsto rv[\varsigma]$ in L it holds that $\mathcal{C} \vdash \varsigma : \Gamma(x)$. This judgment essentially checks that dynamically typed variables have either dynamic or public labels. A **heap** μ is well-typed, written $\vdash \mu$, if for all field values it contains, the value label has the type of the corresponding field declaration.

The type soundness and non-inference lemmas require a typing judgment for execution contexts, $\gamma, \ell, g \vdash E : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E} | \ell', g'$. The judgment describes the typing environment of the execution context, and, given its program counter labels ℓ, g, it defines the program counter labels ℓ', g' that are active at its hole. For example, starting with labels ℓ, g, the context $\mathbf{cx}\,[\mathcal{L}, \mathcal{G}, \mathcal{R}, \ell']\{\langle\rangle\}$ has the labels ℓ' and $\ell' \sqcup g$ active at its hole. The constraints, effect and environments that make up a context typing are the same as for statement typing. The technical report contains the corresponding rules. They are unsurprising; the premises for each context (e.g. **cx**) are similar to those of the statements that enter the context (**if**).

A typed configuration $E(s)/L/\,\mu$ combines the typings of its individual components. Thus we write $\gamma, \ell, g \vdash E(s)/L/\,\mu : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E} | \ell', g'$ if (i) $\Gamma_1, \mathcal{C} \vdash L$ (ii) $\vdash \mu$ and $\gamma, \ell, g \vdash E : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E} | \ell', g'$, (iii) $\gamma \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}$ and $\mathcal{C} \vdash \ell : \gamma$ (iv) $\ell \leq g$. As usual for a gradually typed system, a well typed LJGS program guarantees a refined progress property. While (security related) run-time errors in static code are ruled out, a program may run into a *dynamically stuck* configuration where dynamic code goes wrong.

▶ **Definition 8** (Dynamically stuck)**.** A well-typed configuration $\gamma, \ell, g \vdash E(s)/L/\,\mu : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E} | \ell', g'$ is *dynamically stuck* iff it attempts either: (i) an insecure value cast from

dynamic to static, $s = (var = (a \Leftarrow \star)x)$, $a \in \{\bullet, A\}$, $L[x] = rv[\mathrm{D}(A')]$, and $a \not\leq A$, or (ii) an insecure context cast from dynamic to static $s = ((\star \Rrightarrow a)\{s'\})$, $\ell' = \mathrm{D}(PC)$, $a \in \{\bullet, PC\}$and $PC \not\sqsubseteq A$

The progress result for LJGS states that well typed programs that are not able to make a reduction step are either **done**, or dynamically stuck. The preservation result is standard: a well typed LJGS configuration that can be reduced yields another well-typed configuration with possibly more permissible constraints and effects. The precise definitions for progress and preservation are given in the technical report.

8.2 Non-Interference

The non-interference theorem for LJGS states that methods run under *low-equivalent environments*, that is, environments that an attacker cannot distinguish, produce *low-equivalent results*. First we define the notion of *low-equivalence* and subsequently state the non-interference theorem. In the following, we assume that *low* is the upper bound of security levels that an attacker can observe. We refer to a security level A as *high* if $A \not\sqsubseteq low$.

▶ **Definition 9** (Low-equivalent values, objects, heaps and stack-frames)**.** Let B be a (partial) bijective mapping on heap references (*oref*s). **Two values** $rv_1[\varsigma], rv_2[\varsigma]$ are equivalent under B, written $rv_1[\varsigma] =_B rv_2[\varsigma]$ iff either $rv_{1/2}$ are equal integers, or $rv_1 \in \mathrm{dom}\,(B)$ and $B(rv_1) = rv_2$. **Two objects** obj_1, obj_2 are low-equivalent under B, written $obj_1 =_{B,low} obj_2$, if for all fields F where $\mathrm{getfield}\,(F, obj_1) \neq_B \mathrm{getfield}\,(F, obj_2)$ it holds that either (i) $\mathrm{fsec}\,(F) = A$ and $A \not\sqsubseteq low$, or (ii) $\mathrm{fsec}\,(F) = \star$, $\mathrm{getfield}\,(F, obj_1) = rv_1[\mathrm{D}(A_1)]$, $\mathrm{getfield}\,(F, obj_2) = rv_1[\mathrm{D}(A_2)]$, and $A_1 \not\sqsubseteq low$ and $A_2 \not\sqsubseteq low$. **Two heaps** μ_1, μ_2 are low-equivalent, written $\mu_1 \equiv_{B,low} \mu_2$ if (i) $\mathrm{dom}\,(\mu_1) \supseteq \mathrm{dom}\,(B)$ and $\mathrm{dom}\,(\mu_2) \supseteq \mathrm{dom}\,(B^{-1})$, and (ii) the objects stored at the references listed in B are low-equivalent with respect to B. **Two environments** L_1, L_2 are low-equivalent with respect to environment Γ, solution θ, and bijection B, written $L_1 \equiv_{\Gamma,\theta,B,low} L_2$ if $\mathrm{dom}\,(L_1) = \mathrm{dom}\,(L_2)$, $L_1[\mathbf{this}] = L_2[\mathbf{this}]$, and for all $var \in \mathrm{dom}\,(L_1)$ either (i) $L_1[var] = rv_1[\varsigma_1], L2[var] = rv2[va2]]$, and $rv_1 =_B rv_2$, (ii) $\theta(\Gamma(var)) = A$, and $A \not\sqsubseteq low$, or (iii) $\theta(\Gamma(var)) = \star$ and $\varsigma_1 = \mathrm{D}(A_1)$, $\varsigma_2 = \mathrm{D}(A_2)$, and $A_1 \not\sqsubseteq low$ and $A_2 \not\sqsubseteq low$

▶ **Theorem 10** (Non-interference)**.** *Let $E(s)/L_1/\mu_1$, $E(s)/L_2/\mu_2$ be two configurations with typings $\gamma, \ell, g \vdash E(s)/L_i/\mu_i : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}|\ell', g'$, $(i \in \{1, 2\})$ and solution $\theta \models \mathcal{C}$. Let B be a bijection such that $L_1 \equiv_{\Gamma_1,\theta,B,low} L_2$ and $\mu_1 \equiv_{B,low} \mu_2$. Given two executions ℓ; $g \vdash E(s)/L_i/\mu_i \longrightarrow^* \langle \mathbf{done} \rangle / L_i'/\mu_i'$ there exists a bijection $B' \supseteq B$ such that $L_1' \equiv_{\Gamma_2,\theta,B',low} L_2'$ and $\mu_1' \equiv_{B',low} \mu_2'$.*

9 Related Work

There is a large body of prior work on security type systems that ultimately goes back to Denning and Denning's classic paper on information flow security [11]. We focus on discussing works on security type systems and dynamic security enforcement designed for "main-stream" programming languages and defer the reader to the overview articles of Sabelfeld and Myers [24] and Hedin and Sabelfeld [16] for other aspects of language based security.

FlowCaML [22] is an ML dialect supporting static polymorphic security types with security constraints similar to those of LJGS. FlowCaML additionally supports higher-order types and complete type inference. Sun, Banerjee and Naumann describe a modular polymorphic type system for a Java-like, object oriented language [30]. The polymorphic method signatures are comparable to the static fragment of LJGS, but they do not specify a

constraint-based typechecking algorithm. Instead, they apply a standard algorithm to all instances of signatures. Their system additionally supports class definitions with security type parameters and full type inference. Both approaches seem compatible with LJGS and we plan to investigate how they could be adapted for gradual security typing in future work. Barthe et al describe an information flow type system for Java Bytecode and develop a corresponding certified type checker [7]. Their system supports Objects, virtual, monomorphic methods, exceptions and arrays. JIF [21] is an extension to Java with static security types and first-class dynamic security-labels. In JIF, security types may depend on dynamic labels and the interaction of labels and types is verified statically during type checking. In contrast, LJGS does not restrict dynamically labeled values statically but enforces non-interference at run-time. Also, while JIF's dynamic labels may be used to implement some form of ad-hoc dynamic IFC, LJGS aims for the integration of principled dynamic IFC techniques.

LJGS' run-time security enforcement for values with dynamic labels is an adaption of Russo and Sabelfeld's technique for hybrid information flow control [23]. Chandra and Franz [9] present an implementation of a hybrid IFC framework for Java Bytecode that is based on the same principles and closely resembles LJGS' enforcement for dynamic fragments. The updaterefs function that supports dynamic IFC in LJGS can be implemented with off-the-shelf points-to analysis like that proposed by Khedker et al [18]. We refer the interested reader to the survey paper of Smaragdakis and Balatsouras [28] for further related work on static points-to analysis. Moore and Chong [20] studied the applicability of static points-to analyses to improve the efficiency of hybrid IFC. Using the points-to information in the hybrid monitor allows their system to infer when heap values do not need tracking anymore. Although their ideas seem compatible with LJGS, our hybrid enforcement settles for an analysis of simple write effects and we rely on explicit static typing for optimization.

Austin and Flanagan propose a series of sound, purely dynamic IFC techniques, the most basic being the no-sensitive-upgrade policy, that do not rely on prior static analysis [3, 4, 5]. These approaches are also compatible with LJGS' type system.

Recently, Bedford et al proposed a type system for a simple imperative core language that also includes types for entities with statically unknown security levels [8]. Their work focuses on inferring program points for run-time instrumentation based on the typing results. In contrast, LJGS' focus is on giving the programmer explicit control over the boundaries between static and dynamic checking.

The original work on gradual typing [32, 19, 27, 34] focuses on simple types with extensions like refinement predicates, polymorphism [1], and union types [33]. More recently, researchers started to gradualize type systems that check properties unrelated to the structure of values, like type annotations [14], ownership [25], typestate [35], and session types [31]. Gradual security type systems also fall into this category. Disney and Flanagan study gradual security types for a pure lambda calculus [12] and we describe MLGS, a system for a calculus with ML style references, in our prior work [13]. Compared with LJGS treatment of object fields, MLGS treats mutable references in a more liberal way because it admits casts between reference types with static and dynamic content. However, this liberality comes at the cost of requiring pervasive run-time labelling even for static values and it blurs the separation of static and dynamic code, which, although sound, runs contrary the execution model and design goal of LJGS.

10 Conclusion and Future Work

LJGS is a sequential Java core calculus with gradual security typing. The calculus strictly separates statically verified and dynamically checked code which enables running statically checked code without run-time security labels. Methods may have polymorphic security signatures that accept static or dynamic arguments.

There are several avenues for future work. We are currently implementing the type checking and run-time enforcement for (sequential) Java based on the principles of LJGS. A type checker is already available as an accompanying artifact to this paper but the run-time instrumentation is still work in progress. With this implementation we want to investigate the practicality of the type system for realistic applications and to evaluate different compilation- and execution strategies for dynamic code. We also want to extend the system with security type parameters for classes and methods, and type inference.

References

1 Amal Ahmed, Robert Bruce Findler, Jacob Matthews, and Philip Wadler. Blame for all. In *Proceedings of the 1st Workshop on Script to Program Evolution*, pages 1–13, Genova, Italy, 2009. ACM.

2 Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.

3 Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In Stephen Chong and David A. Naumann, editors, *PLAS*, pages 113–124, Dublin, Ireland, June 2009. ACM.

4 Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 3:1–3:12, New York, NY, USA, 2010. ACM.

5 Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In John Field and Michael Hicks, editors, *Proc. 39th ACM Symp. POPL*, pages 165–178, Philadelphia, USA, January 2012. ACM Press.

6 Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *CSFW*, pages 253–, Cape Breton, Nova Scotia, Canada, June 2002. IEEE Computer Society.

7 Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference Java bytecode verifier. *Mathematical Structures in Computer Science*, 23(5):1032–1081, 2013.

8 Andrew Bedford, Josée Desharnais, Théophane G. Godonou, and Nadia Tawbi. Enforcing information flow by combining static and dynamic analysis. In Jean Luc Danger, Mourad Debbabi, Jean-Yves Marion, Joaquín García-Alfaro, and A. Nur Zincir-Heywood, editors, *Foundations and Practice of Security - 6th International Symposium, FPS 2013, La Rochelle, France, October 21-22, 2013, Revised Selected Papers*, volume 8352 of *LNCS*, pages 83–101, La Rochelle, France, October 2013. Springer.

9 Deepak Chandra and Michael Franz. Fine-grained information flow analysis and enforcement in a Java virtual machine. In *23rd Annual Computer Security Applications Conference (ACSAC 2007)*, pages 463–475, Miami Beach, Florida, USA, December 2007. IEEE Computer Society.

10 Dorothy Denning. A lattice model of secure information flow. *Comm. ACM*, 19(5):236–242, 1976.

11 Dorothy Denning and Peter Denning. Certification of programs for secure information flow. *Comm. ACM*, 20(7):504–513, 1977.

12 Tim Disney and Cormac Flanagan. Gradual information flow typing. In *STOP*, 2011.

13 Luminous Fennell and Peter Thiemann. Gradual security typing with references. In Véronique Cortier and Anupam Datta, editors, *CSF*, pages 224–239, New Orleans, LA, USA, 2013. IEEE.

14 Luminous Fennell and Peter Thiemann. Gradual typing for annotated type systems. In Zhong Shao, editor, *ESOP'14*, volume 8410 of *Lecture Notes in Computer Science*, pages 47–66, Grenoble, France, April 2014. Springer.

15 Manuel Geffken and Peter Thiemann. Side effect monitoring for Java using bytecode rewriting. In Joanna Kolodziej and Bruce R. Childers, editors, *PPPJ '14*, pages 87–98, Cracow, Poland, September 2014. ACM.

16 Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In *2011 Marktoberdorf Summer School*. IOS Press, 2011.

17 Sebastian Hunt and David Sands. On flow-sensitive security types. In Simon Peyton Jones, editor, *Proc. 33rd ACM Symp. POPL*, pages 79–90, Charleston, South Carolina, USA, January 2006. ACM Press.

18 Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *ACM TOPLAS*, 30(1), 2007.

19 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In Matthias Felleisen, editor, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 3–10, Nice, France, January 2007. ACM Press.

20 Scott Moore and Stephen Chong. Static analysis for efficient hybrid information-flow control. In *CSF 2011*, pages 146–160, Cernay-la-Ville, France, June 2011. IEEE Computer Society.

21 Andrew C. Myers. JFlow: Practical mostly-static information flow control. In Alexander Aiken, editor, *Proc. 26th ACM Symp. POPL*, pages 228–241, San Antonio, Texas, USA, January 1999. ACM Press.

22 François Pottier and Vincent Simonet. Information flow inference for ML. *ACM TOPLAS*, 25(1):117–158, January 2003.

23 Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *CSF*, pages 186–199. IEEE Computer Society, 2010.

24 Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

25 Ilya Sergey and Dave Clarke. Gradual ownership types. In *21th European Symposium on Programming (ESOP 2012)*, Tallinn, Estonia, April 2012. Springer.

26 Jeremy Siek and Walid Taha. Gradual typing for objects. In Erik Ernst, editor, *21st ECOOP*, volume 4609 of *LNCS*, pages 2–27, Berlin, Germany, July 2007. Springer.

27 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.

28 Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.

29 Rok Strnisa and Matthew J. Parkinson. Lightweight Java. *Archive of Formal Proofs*, 2011, 2011.

30 Qi Sun, Anindya Banerjee, and David A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In Roberto Giacobazzi, editor, *SAS 2004*, volume 3148 of *LNCS*, pages 84–99, Verona, Italy, August 2004. Springer.

31 Peter Thiemann. Gradual typing for session types. In Emilio Tuosto and Matteo Maffeis, editors, *TGC*, volume 8902 of *LNCS*, pages 144–158, Rome, Italy, September 2014. Springer.

32 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium, DLS 2006*, pages 964–974, Portland, Oregon, USA, 2006. ACM.

33 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In Phil Wadler, editor, *Proc. 35th ACM Symp. POPL*, pages 395–406, San Francisco, CA, USA, January 2008. ACM Press.

34 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Proc. 18th ESOP*, volume 5502 of *LNCS*, pages 1–16, York, UK, March 2009. Springer.

35 Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *ECOOP*, volume 6813 of *LNCS*, pages 459–483, Lancaster, UK, 2011. Springer.

36 Stephan Arthur Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell, Ithaca, NY, USA, 2002.

Formal Language Recognition with the Java Type Checker

Yossi Gil[1] and Tomer Levy[2]

1 **Department of Computer Science, The Technion—Israel Institute of Technology, Haifa, Israel.**
2 **Department of Computer Science, The Technion—Israel Institute of Technology, Haifa, Israel.**

"Java generics are 100% pure syntactic sugar, and do not support meta-programming"[1]

Abstract

This paper is a theoretical study of a practical problem: the automatic generation of Java Fluent APIs from their specification. We explain why the problem's core lies with the expressive power of Java generics. Our main result is that automatic generation is possible whenever the specification is an instance of the set of deterministic context-free languages, a set which contains most "practical" languages. Other contributions include a collection of techniques and idioms of the limited meta-programming possible with Java generics, and an empirical measurement demonstrating that the runtime of the "javac" compiler of Java may be exponential in the program's length, even for programs composed of a handful of lines and which do not rely on overly complex use of generics.

1998 ACM Subject Classification D.3.2: Java, D.3.4 Processors: Parsing, D.3.2 Language classifications: Nonprocedural languages, Specialized application languages, F.4.2 Grammars and Other Rewriting Systems: Classes defined by grammars or automata, Classes defined by resource-bounded automata, F.1.1 Models of Computation: Automata

Keywords and phrases Parser Generators, Generic Programming, Fluent API

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.10

1 Introduction

Ever after their inception[2] *fluent APIs* increasingly gain popularity [20, 28, 31] and research interest [16,29]. In many ways, fluent APIs are a kind of *internal **D**omain **S**pecific **L**anguage*: They make it possible to enrich a host programming language without changing it. Advantages are many: base language tools (compiler, debugger, IDE, etc.) remain applicable, programmers are saved the trouble of learning a new syntax, etc. However, these advantages come at the cost of expressive power; in the words of Fowler: "*Internal DSLs are limited by the syntax and structure of your base language.*"[3]. Indeed, in languages such as C++ [37], fluent APIs often make extensive use of operator overloading (examine, e.g., Ara-Rat [23]), but this capability is not available in Java [4].

[1] Found on stackoverflow: `http://programmers.stackexchange.com/questions/95777/generic-programming-how-often-is-it-used-in-industry`
[2] `http://martinfowler.com/bliki/FluentInterface.html`
[3] M. Fowler, *Language Workbenches: The Killer-App for Domain Specific Languages?*, 2005 `http://www.martinfowler.com/articles/languageWorkbench.html#InternalDsl`

```
from("direct:a").choice()
    .when(header("foo").isEqualTo("bar"))
      .to("direct:b")
    .when(header("foo").isEqualTo("cheese"
     ))
      .to("direct:c")
    .otherwise()
      .to("direct:d");
```

(a) Apache Camel

```
create
  .select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME, count
     ())
  .from(AUTHOR)
  .join(BOOK).on(AUTHOR.ID.equal(BOOK.AUTHOR_ID))
  .where(BOOK.LANGUAGE.eq("DE"))
  .and(BOOK.PUBLISHED.gt(date("2008-01-01")))
  .groupBy(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
  .having(count().gt(5))
  .orderBy(AUTHOR.LAST_NAME.asc().nullsFirst())
  .limit(2)
  .offset(1);
```

(b) jOOQ

Figure 1 Two examples of Java fluent API.

Despite this limitation, fluent APIs in Java can be rich and expressive, as demonstrated in Figure 1 showing use cases of the DSL of Apache Camel [27] (open-source integration framework), and that of jOOQ[4], a framework for writing SQL in Java, much like Linq [33].

Other examples of fluent APIs in Java are abundant: jMock [20], Hamcrest[5], EasyMock[6], jOOR[7], jRTF[8] and many more.

1.1 A Type Perspective on Fluent APIs

Figure 1(B) suggests that jOOQ imitates SQL, but, is it possible at all to produce a fluent API for the entire SQL language, or XPath, HTML, regular expressions, BNFs, EBNFs, etc.? Of course, with no operator overloading it is impossible to fully emulate tokens; method names though make a good substitute for tokens, as done in "`.when(header(foo).isEqualTo("bar")).`" (Figure 1). The questions that motivate this research are:

- Given a specification of a DSL, determine whether there exists a fluent API that can be made for this specification?
- In the cases that such fluent API is possible, can it be produced automatically?
- Is it feasible to produce a *compiler-compiler* such as Bison [17] to convert a language specification into a fluent API?

Inspired by the theory of formal languages and automata, this study explores what can be done with fluent APIs in Java.

Consider some fluent API (or DSL) specification, permitting only certain call chains and disallowing all others. Now, think of the formal language that defines the set of these permissible chains. We prove that there is always a Java type definition that *realizes* this fluent definition, provided that this language is *deterministic context-free*, where

- In saying that a type definition *realizes* a specification of fluent API, we mean that call chains that conform with the API definition compile correctly, and, conversely, call chains that are forbidden by the API definition do not type-check, resulting in an appropriate compiler error.

[4] http://www.jooq.org
[5] http://hamcrest.org/JavaHamcrest/
[6] http://easymock.org/
[7] https://github.com/jOOQ/jOOR
[8] https://github.com/ullenboom/jrtf

- Roughly speaking, deterministic context-free languages are those context-free languages that can be recognized by an LR parser[9] [2].
  An important property of this family is that none of its members is ambiguous. Also, it is generally believed that most practical programming languages are deterministic context-free.

A problem related to that of recognizing a formal language, is that of parsing, i.e., creating, for input which is within the language, a parse tree according to the language's grammar. In the domain of fluent APIs, the distinction between recognition and parsing is in fact the distinction between compile time and runtime. Before a program is run, the compiler checks whether the fluent API call is legal, and code completion tools will only suggest legal extensions of a current call chain.

In contrast, a parse tree can only be created at runtime. Some fluent API definitions create the parse-tree iteratively, where each method invocations in the call chain adding more components to this tree. However, it is always possible to generate this tree in "batch" mode: This is done by maintaining a *fluent-call-list* which starts empty and grows at runtime by having each method invoked add to it a record storing the method's name and values of its parameters. The list is completed at the end of the fluent-call-list, at which point it is fed to an appropriate parser that converts it into a parse tree (or even an AST).

1.2 Contribution

The answers we provide for the three questions above are:

1. If the DSL specification is that of a deterministic context-free language, then a fluent API exists for the language, but we do not know whether such a fluent API exists for more general languages.
   Recall that there are universal cubic time parsing algorithms [13,18,40] which can parse (and recognize) any context-free language. What we do not know is whether algorithms of this sort can be encoded within the framework of the Java type system.
2. There exists an algorithm to generate a fluent API that realizes any deterministic context-free languages. Moreover, this fluent API can create at runtime, a parse tree for the given language. This parse tree can then be supplied as input to the library that implements the language's semantics.
3. Unfortunately, a general purpose compiler-compiler is not yet feasible with the current algorithm.
   - One difficulty is usual in the fields of formal languages: The algorithm is complicated and relies on modules implementing complicated theoretical results, which, to the best of our knowledge, have never been implemented.
   - Another difficulty is that a certain design decision in the implementation of the standard `javac` compiler is likely to make it choke on the Java code generated by the algorithm.

Other concrete contributions made by this work include

- the understanding that the definition of fluent APIs is analogous to the definition of a formal language.
- a lower bound (deterministic pushdown automata) on the theoretical "computational complexity" of the Java type system.

[9] The "L" means reading the input left to right; the "R" stands for rightmost derivation

- an algorithm for producing a fluent API for deterministic context-free languages.
- a collection of generic programming techniques, developed towards this algorithm.
- a demonstration that the runtime of Oracle's `javac` compiler may be exponential in the program size.

1.3 Related Work

It has long been known that C++ templates are Turing complete in the following precise sense:

▶ Proposition 1. For every Turing machine, m, there exists a C++ program, C_m such that compilation of C_m terminates if and only if Turing-machine m halts. Furthermore, program C_m can be effectively generated from m [25].

Intuitively, this is due to the fact that templates in C++ feature both recursive invocation and conditionals (in the form of "*template specialization*").

In the same fashion, it should be mundane to make the judgment that Java's generics are not Turing-complete since they offer no conditionals. Still, even though there are time complexity results regarding type systems in functional languages, we failed to find similar claims for Java.

Specialization, conditionals, `typedef`s and other features of C++ templates, gave rise to many advancements in template/generic/generative programming in the language [5, 7, 15, 34], including e.g., applications in numeric libraries [38, 39], symbolic derivation [22] and a full blown template library [1].

Garcia et al. [21] compared the expressive power of generics in half a dozen major programming languages. In several ways, the Java approach [11] did not rank as well as others.

Not surprisingly, work on meta-programming using Java generics, research concentrating on other means for enriching the language, most importantly annotations [36].

The work on SugarJ [19] is only one of many other attempts to achieve the embedded DSL effect of fluent APIs by language extensions.

Suggestions for semi-automatic generation can be found in the work of Bodden [10] and on numerous locations in the web. None of these materialized into an algorithm or analysis of complexity. However, there is a software artifact (fluflu[10]) that automatically generates a fluent API that obeys the transitions of a given finite automaton.

Outline. Section 2 is a brief reminder of method chaining, and fluent APIs, accompanied a discussion of how this work is related to type states. It is followed by a similar reminder of context-free languages, pushdown automata, and such in Section 3. Based on the vocabulary established this far, the main result is stated in Section 4.

Towards the proof in Section 7, Section 5 shows idioms and techniques for encoding computation with the Java type-checker. Section 6 makes use of these for encoding "jump-stack", a non-trivial data-structure, which is used, with suitable modifications, in the proof.

In Section 9, we discuss the challenges in translating the proof into a compiler-compiler for fluent APIs. In particular, this section demonstrates our claim (that may be surprising to some) that the standard Java compiler may spend an exponential time on compiling rather simple programs. Section 10 concludes with directions for further research.

[10] `https://github.com/verhas/fluflu`

```
String time(int hours, int minutes, int seconds)
      {
  final StringBuilder sb = new StringBuilder();
  sb.append(hours);
  sb.append(':');
  sb.append(minutes);
  sb.append(':');
  sb.append(seconds);
  return sb.toString();
}
```

(a) before

```
String time(int hours, int minutes, int seconds)
      {
    return new StringBuilder()
      .append(hours).append(':')
      .append(minutes).append(':')
      .append(seconds)
      .toString();
}
```

(b) after

Figure 2 Recurring invocations of the pattern "invoke function on the same receiver", before, and after method chaining.

2 Method Chaining, Fluent APIs, and, Type States

The pattern "invoke function on variable **sb**", specifically with a function named **append**, occurs six times in the code in Figure 2(a), designed to format a clock reading, given as integers hours, minutes and seconds.

Some languages, e.g., SMALLTALK [24] offer syntactic sugar, called *cascading*, for abbreviating this pattern. *Method chaining* is a "programmer made" syntactic sugar serving the same purpose: If a method f returns its receiver, i.e., **this**, then, instead of the series of two commands: `o.f(); o.g();`, clients can write only one: `o.f().g();`. Figure 2(b) is the method chaining (also, shorter and arguably clearer) version of Figure 2(a). It is made possible thanks to the designer of class **StringBuilder** ensuring that all overloaded variants of **append** return their receiver.

The distinction between *fluent API* and method chaining is the identity of the receiver: In method chaining, all methods are invoked on the same object, whereas in fluent API the receiver of each method in the chain may be arbitrary. Fluent APIs are more interesting for this reason. Consider, e.g., the following JAVA code fragment (drawn from JMock [20])

```
allowing(any(Object.class)).method("get.*").withNoArguments();
```

Let the return type of function **allowing** (respectively **method**) be denoted by τ_1 (respectively τ_2). Then, the fact that $\tau_1 \neq \tau_2$ means that the set of methods that can be placed after the dot in the partial call chain `allowing(any(Object.class)).` is not necessarily the same set of methods that can be placed after the dot in the partial call chain

```
allowing(any(Object.class)).method("get.*").
```

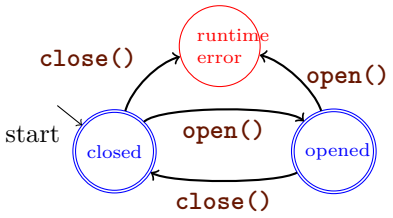
This distinction makes it possible to design expressive and rich fluent APIs, in which a sequence of "chained" calls is not only readable, but also robust, in the sense that the sequence is type correct only when it makes sense semantically.

There is a large body of research on *type-states* (See e.g., review articles such as [3, 9]). Informally, an object that belongs to a certain type, has type-states, if not all methods defined in this object's class are applicable to the object in all states it may be in. As it turns out, objects with type states are quite frequent: a recent study [8] estimates that about 7.2% of JAVA classes define protocols, that can be interpreted as type-state.

In a sense, type states define the "language" of the protocol of an object. The protocol of the type-state **Box** class defined in Figure 3 admits the chain `new Box().open().close()` but not the chain `new Box().open().open()`.

	`open()`	`close()`
"closed"	*become "open"*	*runtime error*
"open"	*runtime error*	*become "closed"*

(a) Definition by table



(b) Definition by DFA

Figure 3 Fluent API of a box object, defined by a DFA and a table.

As mentioned above, tools such as fluflu realize type-state based on their finite automaton description. Our approach is a bit more expressive: examine the language L defined by the type-state, e.g., in the box example,

$$L = \big(\texttt{.open().close()}\big)^*\big(\texttt{.open()} \mid \epsilon\big).$$

If L is deterministic context-free, a fluent API can be made for it.

To make the proof concrete, consider this example of fluent API definition: An instance of class `Box` may receive two method invocations: `open()` and `close()`, and can be in either "open" or "closed" state. Initially the instance is "closed". Its behavior henceforth is defined by Figure 3.

To realize this definition, we need a type definition by which `new Box().open().close()`, more generally blue, or accepting states in the figure, type-check. Conversely, with this type definition, compile time type error should occur in `new Box().close()`, and, more generally, in the red state.

Some skill is required to make this type definition: proper design of class `Box`, perhaps with some auxiliary classes extending it, an appropriate method definition here and there, etc.

3 Context-Free Languages and Pushdown Automata: Reminder and Terminology

Notions discussed here are probably common knowledge (see e.g., [26, 32] for a text book description, or [6] for a scientific review). The purpose here is to set a unifying common vocabulary.

Let Σ be a finite alphabet of *terminals* (often called input characters or tokens). A *language* over Σ is a subset of Σ^*. Keep Σ implicit henceforth.

A ***N****ondeterministic* ***P****ushdown* ***A****utomaton* (NPDA) is a device for language recognition, made of a nondeterministic finite automaton and a stack of unbounded depth of (stack) *elements*. A NPDA begins execution with a single copy of the initial element on the stack. In each step, the NPDA examines the next input token, the state of the automaton, and the top of the stack. It then pops the top element from the stack, and nondeterministically chooses which actions of its transition function to perform: Consuming the next input token, moving to a new state, or, pushing any number of elements to the stack. Actually, any combination of these actions may be selected.

The language recognized by a NPDA is the set of strings that it accepts, either by reaching an accepting state or by encountering an empty stack.

A ***C**ontext-**F**ree **G**rammar*(CFG) is a formal description of a language. A CFG G has three components: Ξ a set of *variables* (also called nonterminals), a unique *start variable* $\xi \in \Xi$, and a finite set of (production) *rules*. A rule $r \in G$ describes the derivation of a variable $\xi \in \Xi$ into a string of *symbols*, where symbols are either terminals or variables. Accordingly, rule $r \in G$ is written as $r = \xi \to \beta$, where $\beta \in (\Sigma \cup \Xi)^*$. This description is often called BNF. The *language* of a CFG is the set of strings of terminals (and terminals only) that can be derived from the start symbol, following any sequence of applications of the rules. CFG languages make a proper superset of regular languages, and a proper subset of "context-sensitive" languages [26].

The expressive power of NPDAs and BNFs is the same: For every language defined by a BNF, there exists a NPDA that recognizes it. Conversely, there is a BNF definition for any language recognized by some NPDA.

NPDAs run in exponential deterministic time. A more sane, but weaker, alternative is found in LR(1) parsers, which are deterministic linear time and space. Such parsers employ a stack and a finite automaton structure, to parse the input. More generally, LR(k) parsers, $k > 1$, can be defined. These make their decisions based on the next k input character, rather than just the first of these. General LR(k) parsers are rarely used, since they offer essentially the same expressive power[11], at a greater toll on resources (e.g., size of the automaton). In fact, the expressive power of LR(k), $k \geq 1$ parsers, is that of "***D**eterministic **P**ushdown **A**utomaton*" (DPDA), which are similar to NPDA, except that their conduct is deterministic.

▶ Definition 1 (Deterministic Pushdown Automaton). *A deterministic pushdown automaton (DPDA) is a quintuple* $\langle Q, \gamma, q_0, A, \delta \rangle$ *where* Q *is a finite set of* states, γ *is a finite* set of elements, $q_0 \in Q$ *is the initial state, and* $A \subseteq Q$ *is the* set of accepting states *while* δ *is the* partial state transition function $\delta : Q \times \gamma \times (\Sigma \cup \{\epsilon\}) \nrightarrow Q \times \gamma^*$.

A DPDA begins its work in state q_0 *with a single designated stack element residing on the stack. At each step, the automaton examines: the current state* $q \in Q$*, the element* $\gamma \in \gamma$ *at the top of the stack, and* σ*, the next input token, Based on the values of these, it decides how to proceed:*

1. *If* $q \in A$ *and the input is exhausted, the automaton accepts the input and stops.*
2. *Suppose that* $\delta(q, \gamma, \epsilon) \neq \bot$ *(in this case, the definition of a DPDA requires that* $\delta(q, \gamma, \sigma') = \bot$ *for all* $\sigma' \in \Sigma$*), and let* $\delta(q, \gamma, \epsilon) = (q', \zeta)$*. Then the automaton pops* γ *and pushes the string of stack elements* $\zeta \in \gamma^*$ *into the stack.*
3. *If* $\delta(q, \gamma, \sigma) = (q', \zeta)$*, then the same happens, but the automaton also irrevocably consumes the token* σ*.*
4. *If* $\delta(q, \gamma, \epsilon) = \delta(q, \gamma, \sigma) = \bot$ *the automaton rejects the input and stops.*

A *configuration* is the pair of the current state and the stack contents. Configurations represent the complete information on the state of an automaton at any given point during its computation. A *transition* of a DPDA takes it from one configuration to another. Transitions which do not consume an input character are called *ϵ-transitions*.

As mentioned above, NPDA languages are the same as CFG languages. Equivalently, *DCFG languages* (deterministic context-free grammar languages) are context-free languages that are recognizable by a DPDA. The set of DCFG languages is still a proper superset of regular languages, but a proper subset of CFG languages.

[11] they recognize the same set of languages [30].

4 Statement of the Main Result

Let java be a function that translates a terminal $\sigma \in \Sigma$ into a call to a uniquely named function (with respect to σ). Let java(α), be the function that translates a string $\alpha \in \Sigma^*$ into a fluent API call chain. If $\alpha = \sigma_1 \cdots \sigma_n \in \Sigma^*$, then

$$\mathsf{java}(\alpha) = \mathsf{java}(\sigma_1)\texttt{().}\cdots\texttt{.}\mathsf{java}(\sigma_n)\texttt{()}$$

For example, when $\Sigma = \{a, b, c\}$ let java(a) = `a`, java(b) = `b`, and, java(c) = `c`. With these,

$$\mathsf{java}(caba) = \texttt{c().a().b().a()}$$

▶ **Theorem 1.** *Let A be a DPDA recognizing a language $L \subseteq \Sigma^*$. Then, there exists a* Java *type definition, J_A for types* `L`, `A` *and other types such that the* Java *command*

$$\texttt{L } \ell \texttt{ = A.build.}\mathbf{\mathsf{java}}(\alpha)\texttt{.\$();} \tag{1}$$

type checks against J_A if an only if $\alpha \in L$. Furthermore, program J_A can be effectively generated from A.

Equation 1 reads: starting from the `static` field `build` of `class` `A`, apply the sequence of call chain java(α), terminate with a call to the ending character `$()` and then assign to newly declared Java variable ℓ of type `L`.

The proof of the theorem is by a scheme for encoding in Java types the pushdown automaton $A = A(L)$ that recognizes language L. Concretely, the scheme assigns a type $\tau(c)$ to each possible configuration c of A. Also, the type of `A.build` is $\tau(c_0)$, where c_0 is the initial configuration of A,

Further, in each such type the scheme places a function $\sigma()$ for every $\sigma \in \Sigma$. Suppose that A takes a transition from configuration c_i to configuration c_j in response to an input character σ_k. Then, the return type of function σ_k`()` in type $\tau(c_i)$ is type $\tau(c_j)$.

With this encoding the call chain in Equation 1 mimics the computation of A, starting at c_0 and ending with rejection or acceptance. The full proof is in Section 7.

Since the depth of the stack is unbounded, the number of configurations of A is unbounded, and the scheme must generate an infinite number of types. Genericity makes this possible, since a generic type is actually device for creating an unbounded number of types.

There are several, mostly minor, differences between the structure of the Java code in Equation 1 and the examples of fluent API we saw above,e.g., in Figure 1:

Prefix, i.e., the starting `A.build` variable. All variables and functions of Java are defined within a class. Therefore, a call chain must start with an object (`A.build` in Equation 1) or, in case of `static` methods, with the name of a class. In fluent API frameworks this prefix is typically eliminated with appropriate `import` statements.

If so desired, the same can be done by our type encoding scheme: define all methods in type $\tau(c_0)$ as `static` and `import static` these.

Suffix, i.e., the terminal `.$()` call. In order to know whether $\alpha \in L$ the automaton recognizing L must know when α is terminated.

With a bit of engineering, this suffix can also be eliminated. One way of doing so is by defining type `L` as an `interface`, and by making all types $\tau(c)$, c is an accepting configuration, as subtype of `L`.

Parameterized methods. Fluent API frameworks support call chains with phrases such as:

- "`.when(header(foo).isEqualTo("bar")).`",
- "`.and(BOOK.PUBLISHED.gt(date("2008-01-01"))).`", and,

- "`.allowing(any(Object.class)).`".

while our encoding scheme assumes methods with no parameters.

Methods with parameters contribute to the user experience and readability of fluent APIs but their "computational expressive power"' is the same. In fact, extending Theorem 1 to support these requires these conceptually simple steps

1. Define the structure of parameters to methods with appropriate fluent API, which may or may not be, the same as the fluent API of the outer chain, or the fluent API of parameters to other methods. Apply the theorem to each of these fluent APIs.
2. If there are several overloaded versions of a method, consider each such version as a distinct character in the alphabet Σ and in the type encoding of the automaton.
3. Add code to the implementation of each method code to store the value of its argument(s) in a record placed at the end of the fluent-call-list.

5 Techniques of Type Encoding

This section presents techniques and idioms of type encoding in JAVA partly to serve in the proof of Theorem 1, and partly to familiarize the reader with the challenges of type encoding.

Let $g : \gamma \nrightarrow \gamma$ be a partial function, from the finite set γ into itself. We argue that g can be represented using the compile-time mechanism of JAVA. Figure 4 encodes such a partial function for $\gamma = \{\gamma_1, \gamma_2\}$, where $g(\gamma_1) = \gamma_2$ and $g(\gamma_2) = \perp$, i.e., $g(\gamma_2)$ is undefined.[12]

The type hierarchy depicted in Figure 4(a) shows five classes: Abstract class γ [13] represents the set γ, final classes γ`1`, γ`2` that extend γ, represent the actual members of the set γ. The remaining two classes are private final class ¤ that stands for an error value, and abstract class γ' that denotes the augmented set $\gamma \cup \{¤\}$. Accordingly, both classes ¤ and γ extend γ'.[14]

The full implementation of these classes is provided in Figure 4(b). This actual code excerpt should be placed as a nested class of some appropriate host class. Import statements are omitted, here and henceforth for brevity.

The use cases in Figure 4(c) explain better what we mean in saying that function g is encoded in the type system: An instance of class γ`1` returns a value of type γ`2` upon method call `g()`, while an instance of class γ`2` returns a value of our `private` error type γ'`.¤` upon the same call.

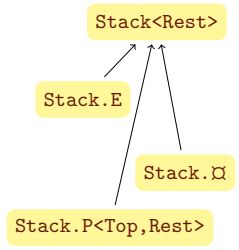
Three recurring idioms employed in Figure 4(b) are:

1. An `abstract` class encodes a set (alternatively, one can use `interfaces`). Abstract classes that extend it encode subsets, while `final` classes encode set members.
2. The interest of frugal management of name-spaces is served by the agreement that if a class X `extends` another class Y, then X is also defined as a `static` member class of Y.
3. Bodies of functions are limited to a single `return null;` command (with interfaces the method body is redundant). This is to stress that at runtime, the code does not carry

[12] Unless otherwise stated, all code excerpts here represent full implementations, and automatically extracted, omitting headers and footers, from JAVA programs that compile correctly with a JAVA 8 compiler.

[13] Remember that JAVA admits Unicode characters in identifier names

[14] The use of short names, e.g., γ instead of $\gamma'.\gamma$, is made possible by an appropriate `import` statement omitted here and henceforth.



(a) type hierarchy

```
public static abstract class Γ′ {
  private static abstract class ¤ extends Γ′ {
    // Empty private class, cannot be used by clients.
    private ¤() { /* Private constructor hinders extension by clients */
      }
  }
  public  static abstract class γ extends γ′ {
    public abstract γ′ g();
    public static final class γ1 extends γ {
      // Covariant return type in overriding:
      @Override public γ2 g() { return null; }
    }
    public static final class γ2 extends γ {
      // Covariant return type in overriding:
      @Override  public γ′.¤ g() { return null; }
    }
  }
}
```

(b) implementation

```
public static void five_use_cases_of_function_g() {
  γ2 _1 = new γ1().g(); // ✓
  γ1 _2 = new γ2().g(); // ✗ type mismatch
  γ′.¤ _3 = new γ2().g(); // ✗ class ¤ is private
  γ′ _4 = new γ2().g(); // ✓
  _4.g(); // ✗ g() undefined in type γ′
}
```

(c) use cases

Figure 4 Type encoding of partial function $g : \gamma \nrightarrow \gamma$, defined by $\gamma = \{\gamma_1, \gamma_2\}$, $g(\gamma_1) = \gamma_2$ and $g(\gamma_2) = \bot$.

out any useful or interesting computation, and the class structure is solely for providing compile-time type checks. [15]

Having seen how inheritance and overriding make possible the encoding of unary functions, we turn now to encoding higher arity functions. With the absence of multi-methods, other techniques must be used.

Consider the partial binary function $f : R \times S \nrightarrow \gamma$, defined by

$$\begin{array}{lll} R = \{r_1, r_2\} & f(r_1, s_1) = \gamma_1 & f(r_2, s_1) = \gamma_1 \\ S = \{s_1, s_2\} & f(r_1, s_2) = \gamma_2 & f(r_2, s_2) = \bot \end{array} . \tag{2}$$

A Java type encoding of this definition of function f is in Figure 5(a); use cases are in Figure 5(b).

As the figure shows, to compute $f(r_1, s_1)$ at compile time we write `f.r1().s1()`. Also, the fluent API call chain `f.r2().s2().g()` results in a compile time error because

$f(r_2, s_2) = \bot$.

Class `f` in the implementation sub-figure serves as the starting point of the little fluent API defined here. The return type of `static` member functions `r1()` and `r2()` is the respective sub-class of class `R`: The return type of function `r1()` is class `R.r1`; the return type of function `r2()` is class `R.r2`.

[15] A consequence of these idioms is that the augmented class γ' is visible to clients. It can be made `private`. Just move class γ to outside of γ', defying the second idiom.

```
public static abstract class f { // Starting point of fluent API
  public static r1 r1() { return null; }
  public static r2 r2() { return null; }
}
public static abstract class R {
  public abstract γ′ s1();
  public abstract γ′ s2();
  public static final class r1 extends R {
    @Override public γ1 s1() { return null; }
    @Override public γ2 s2() { return null; }
  }
  public static final class r2 extends R {
    @Override public γ2 s1() { return null; }
    @Override public γ′.¤ s2() { return null; }
  }
}
```

(a) implementation (except for classes γ, γ', $\gamma 1$, and $\gamma 2$, found in Figure 4).

```
public static void four_use_cases_of_function_f() {
  γ1 _1 = f.r1().s1(); // ✓ f(r1, s1) = γ1
  γ2 _2 = f.r1().s2(); // ✓ f(r1, s2) = γ2
  γ2 _3 = f.r2().s1(); // ✓ f(r2, s1) = γ2
  f.r2().s2().g(); // ✗ method s2() undefined in type Γ'
}
```

(b) use cases

Figure 5 Type encoding of partial binary function $f : R \times S \nrightarrow \gamma$, where $R = \{r_1, r_2\}$, $S = \{s_1, s_2\}$, and f is specified by $f(r_1, s_1) = \gamma_1$, $f(r_1, s_2) = \gamma_2$, $f(r_2, s_1) = \gamma_1$, and $f(r_2, s_2) = \bot$.

```
interface ID<T extends ID<?>> {
  default T id() { return null; }
}
class A implements ID<A> { /**/ }
abstract class B<Z extends B<?>> implements ID<Z> { /**/ }
class C extends B<C> { /**/ }
```

Figure 6 Covariant return type of function `id()` with Java generics.

Instead of representing set S as a class, its members are realized as methods `s1()` and `s2()` in class `R`. These functions are defined as `abstract` with return type `γ'` in `R`. Both functions are overridden in classes `r1` and `r2`, with the appropriate covariant change of their return type,

It should be clear now that the encoding scheme presented in Figure 5 can be generalized to functions with any number of arguments, provided that the domain and range sets are finite. The encoding of sets of unbounded size require means for creating an unbounded number of types. Genericity can be employed to serve this end.

Figure 6 shows a genericity based recipe for a function whose return type is the same as the receiver's type. This recipe is applied in the figure to classes `A`, `B`, and `C`. In each of these classes, the return type of `id` is, without overriding, (at least) the class itself.

It is also possible to encode with Java generic types unbounded data structures, as demonstrated in Figure 7, featuring a use case of a stack of an *unbounded* depth.

In line 3 of the figure, a stack with five elements is created: These are popped in order (ll.4–7,l.9). Just before popping the last item, its value is examined (l.8). Trying then to pop from an empty stack (l.10), or to examine its top (l.11), ends with a compile time error.

Stack elements may be drawn from some abstract set γ. In the figure these are either class $\gamma 1$ or class $\gamma 2$ (both defined in Figure 4). A call to function γi pushes the type γi into

```
1 public static void use_case_of_stack() {
2   // Create a stack a with five items in it:
3   P<γ1, P<γ1, P<γ2, P<γ1, P<γ1, E>>>>> _1 = Stack.empty.γ1().γ1().γ2().γ1().γ1();
4   P<γ1, P<γ2, P<γ1, P<γ1, E>>>> _2 = _1.pop(); // ✓ Pop one item
5   P<γ2, P<γ1, P<γ1, E>>> _3 = _2.pop(); // ✓ Pop another item
6   P<γ1, P<γ1, E>> _4 = _3.pop(); // ✓ Pop yet another item
7   P<γ1, E> _5 = _4.pop(); // ✓ Pop penultimate item
8   γ1 _6 = _5.top(); // ✓ Examine last item
9   E _7 = _5.pop(); // ✓ Pop last item
10  Stack.¤ _8 = _7.pop(); // ✗ Cannot pop from an empty stack
11  γ′.¤ _9 = _7.top(); // ✗ empty stack has no top element
12 }
```

Figure 7 Use cases of a compile-time stack data structure.

the stack, for $i = 1,$. The expression

`Stack.empty.γ1().γ1().γ2().γ1().γ1()`

represents the sequence of pushing the value γ_1 into an empty stack, followed by γ_1, γ_2, γ_1, and, finally, γ_1. This expression's type is that of variable `_1`, i.e.,

`P<γ1,P<γ1,P<γ2,P<γ1,P<γ1,E»»>`

A recurring building block occurs in this type: generic type `P`, *short for "Push"*, which takes two parameters:

1. the *top* of the stack, always a subtype of γ,
2. the *rest* of the stack, which can be of two kinds:
   a. another instantiation of `P` (in most cases),
   b. non-generic type `E`, *short for "Empty"*, which encodes the empty stack. Note that `E` can only occur at the deepest `P`, encoding a stack with one element, in which the rest is empty.

Incidentally, `static` field `Stack.empty` is of type `E`.

Figure 8(a) gives the type inheritance hierarchy of type `Stack` and its subtypes. Figure 8(b) gives the implementation of these types.

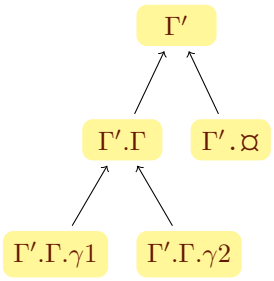
The code in the figure shows that the "rest" parameter of `P` must extend class `Stack`, and that both types `P` and `E` extend `Stack`. Other points to notice are:

- The type at the top of the stack is precisely the return type of `top()`; it is overridden in `P` so that its return type is the first argument of `P`. The return type of `top()` in `E` is the error value $\gamma'.¤$.
- Pushing into the stack is encoded as functions `γ1()` and `γ2()`; the two are overridden with appropriate covariant change of the return type in `P` and `E`.
- Since an empty stack cannot be popped, function `pop()` is overridden in `E` to return the *error* type `Stack.¤`. This type is indeed a kind of a stack, except that each of the four stack functions: `top()`, `push()`, `γ1()`, and, `γ2()`, return an appropriate error type.

In fact, this recursive generic type technique can used to encode S-expressions: In the spirit of Figure 8, the idea is to make use of a `Cons` generic type with covariant `car()` and `cdr()` methods.

A standard technique of template programming in C++ is to encode conditionals with template specialization. Since Java forbids specialization of generics, in lieu we use covariant overloading of function return type (e.g., the return type of `s2()` in Figure 5 and the return type of `top()` in Figure 8).

Figure 9 shows that a similar covariant change is possible in extending a generic type. The type of the parameter `M` to `Heap` is "`? extends Mammals`". This type is specialized as



(a) type hierarchy

```
public static abstract class Stack<Rest extends Stack<?>> {
  public abstract γ′ top();
  public abstract Rest pop();
  public abstract Stack<?> γ1();   // Push type Γ′.Γ.γ1
  public abstract Stack<?> γ2();   // Push type Γ′.Γ.γ2
  public static class P<Top extends γ, Rest extends Stack<?>>
    extends Stack<Rest> { // Type of a non−empty stack:
      @Override public Top top() { return null; }
      @Override public Rest pop() { return null; }
      @Override public P<γ1, P<Top,Rest>> γ1() { return null; }
      @Override public P<γ2, P<Top,Rest>> γ2() { return null; }
  }
  public static final class E
    extends Stack<¤> { // Type of an empty stack
      @Override public γ′.¤ top() { return null; }
      @Override public ¤ pop() { return null; }
      @Override public P<γ1, E> γ1() { return null; }
      @Override public P<γ2, E> γ2() { return null; }
  }
  public static final E empty = null;
  private static final class ¤
    extends Stack<¤> { // Type of pop from empty stack
      @Override public γ′.¤ top() { return null; }
      @Override public ¤ pop() { return null; }
      @Override public ¤ γ1() { return null; }
      @Override public ¤ γ2() { return null; }
  }
}
```

(b) implementation (except for classes γ, γ', $\gamma 1$, and $\gamma 2$, from Figure 4).

Figure 8 Type encoding of an unbounded stack data structure.

```
class Mammals { /* ... */ }
class Heap<M extends Mammals> { /* ... */}
class Whales extends Mammals { /* ... */}
class School<W extends Whales>
  extends Heap<W> { /* ... */}
```

Figure 9 Covariance of parameters to generics.

`School` extends `Heap`: parameter `W` of `School` is of type "`? extends Whales`". Covariant specialization of parameters to generics is yet another idiom for encoding conditionals.

Overloading gives rise to a third idiom for partial emulation of conditionals, as can be seen in Figure 10.

The figure depicts type `Peep` and overloaded versions of `peep()` which together make it possible to extract the top of the stack. The first generic parameter to `Peep` is the top of the stack, the second is the stack itself. Indeed, we see (l.11) that peeping into an empty stack, places a `?` in the first parameter, thanks to the first overloaded version of `peep()` (l.2).

The second overloaded version of `peep()` (ll.3–6) matches against all non-empty stacks. The return type of this version encodes in its first parameter the top of the stack, and in its second parameter, the parameter's type. A use case is in line 9.

Let τ be the type of the top of a given stack. Then, both `top()` and `peep()` can be used to extract τ. There is a subtle difference between the two though: Obtaining τ from `top()` does not make it possible to define variables, function return types, and parameters to functions and generics whose type is τ or depends on it in any way. However, since `Peep` is a type that receives τ as parameter, the body of `Peep` is free to define e.g., functions signature includes on τ, or pass τ further to other generics.

```
1 public static class Peep<γ extends γ′, S extends Stack<? extends Stack<?>>> {}
2 public static Peep<?, E> peep(E _) { return null; } // First overloaded version of peep()
3 public static                                        // Second overloaded version of peep()
4   <Top extends γ, Rest extends Stack<?>> // Two generic parameters
5   Peep<Top, P<Top, Rest>>                // Function return type
6   peep(P<Top, Rest> _) { return null; }  // Function parameters and body
7 public static void peeping_into_a_stack_use_cases() {
8   P<γ2, P<γ1, P<γ2, P<γ1, P<γ2, E>>>>> _1 = Stack.empty.γ2().γ1().γ2().γ1().γ2();
9   Peep<γ2, P<γ2, P<γ1, P<γ2, P<γ1, P<γ2, E>>>>>> _2 = peep(_1);
10  E _3 = Stack.empty;
11  Peep<?, E> _4 = peep(_3);
12 }
```

Figure 10 Peeping into the stack.

```
1 public interface JS<  // 1 + k generic parameters
2     Rest extends JS,              // As a convention, we use JS
3     J_γ1 extends JS,     // with its raw type when no
4     J_γ2 extends JS      // parameters are introduced
5 > {
6   γ′ top();
7   Rest pop();
8   JS γ1();
9   JS γ2();
10  J_γ1 jump_γ1();
11  J_γ2 jump_γ2();
12  interface ¤ extends JS<¤, ¤, ¤> { ... }
13  public interface E extends JS<¤, ¤, ¤> { ... }
14  public static final E empty = null;
15  public interface P<// 2 + k generic arguments:
16      Top extends γ,
17      Rest extends JS,
18      J_γ1 extends JS,
19      J_γ2 extends JS
20  > extends P′<Top, Rest, J_γ1, J_γ2,
21        P<Top, Rest, J_γ1, J_γ2>
22  > { /**/ }
23 }
```

Figure 11 Skeleton of type encoding for the jump-stack data structure.

6 The Jump-Stack Data-Structure

A *jump-stack* is a stack data structure whose elements are drawn from a finite set γ, except that jump-stack supports $\mathsf{jump}(\gamma)$, $\gamma \in \Gamma$ operations (which means "repetitively *pop* elements from the stack up to and including the first occurrence of γ").

Figure 11 shows the skeleton of type-encoding, in parameterized type `JS`, of a jump-stack whose elements are drawn from type γ (Figure 4(b)), i.e., either γ`1` or γ`2`.

Just like `Stack` (Figure 8(b)), `JS` takes a `Rest` parameter encoding the type of a jump-stack after popping. In addition `JS` takes $k = |\gamma|$ type parameters, one for each $\gamma \in \gamma$, which is the type encoding of the jump-stack after a $\mathsf{jump}(\gamma)$ operation. In the figure, there are two such parameters: `J_`γ`1`, and `J_`γ`2`.

Functions defined in `JS` include not only the standard stack operations: `top()`, `pop()`, γ`1()` and γ`2()` (encoding a push of γ_i, $i = 1, 2$, in general, there are k), but also k functions encoding $\mathsf{jump}(\gamma)$, $\gamma \in \gamma$. In our case, these are `jump_`γ`1` and `jump_`γ`2`, which encode $\mathsf{jump}(\gamma_i)$ thanks to their return type being `J_`γi, $i = 1, 2$.

The type hierarchy rooted at `JS` is similar to that of Figure 8(a): Two of the specializations are parameter-less and are almost identical to their `Stack` counterparts: `JS.E` encodes an empty jump-stack; `JS.¤` encodes a jump-stack in error, e.g., after popping from `JS.E`. The body of these two types is omitted here.

Type `JS.P` (lines 16–23 in Figure 11) makes the third specialization of `JS`, encoding a stack with one or more elements. Just like in Figure 6, there are no overridden functions in

```
1 private interface P'<
2 // 2 + k + 1 generic arguments:
3   Top extends γ,
4   Rest extends JS,
5   J_γ1 extends JS,
6   J_γ2 extends JS,
7   Me extends JS
8 > extends JS<Rest, J_γ1, J_γ2> {
9    public Top top();
10   P<γ1, Me, Me, J_γ2> γ1();
11   P<γ2, Me, J_γ2, Me> γ2();
12 }
```

Figure 12 Auxiliary type `P'` encoding succinctly a non-empty jump-stack.

```
public static void jump_stack_use_cases(){
  P<
    γ1,                                  // Top
    P<γ1,P<γ2,E,¤>,P<γ2,E,¤>,E>,E>,// Rest
    P<γ1,P<γ2,E,¤>,P<γ2,E,¤>,E>,E>,// jump(γ1)
    E                                    // jump(γ2)
  > _1 = JS.empty.γ2().γ1().γ1();
  E _2 = _1.jump_γ2();
  P<
    γ1,              // Top
    P<γ2,E,¤>,E>,    // Rest
    P<γ2,E,¤>,E>     // jump(γ1)
    ,E               // jump(γ2)
  > _3 = _1.jump_γ1();
}
```

Figure 13 Use cases for the `JS` type hierarchy.

`JS.P`; it fulfills its duties through the type parameters it takes and the types it passes to `P'` the generic type it extends.

Specifically, `JS.P` takes the same `Top` and `Rest` parameters (ll.17–18) as type `Stack.P`: as well as k additional parameters: `J_γ1` and `J_γ2` (ll.19–20) which are the types encoding the jump-stack after the execution $\mathsf{jump}(\gamma_i)$, $i = 1, 2$. Type `JP.P` passes these four parameters to type `P'` which it extends (l.21). The fifth parameter to `P'` (l.22) is the current incarnation of `P`, i.e., `P<Top, Rest, J_γ1, J_γ2`.

The auxiliary (and `private`) type `P'` itself is depicted in Figure 12. By extending type `JS` and passing the correct `Rest` (respectively, `J_γ1`, `J_γ2`) parameter to it, `P'` inherits correct declaration of function `pop()` (l.8 Figure 11) (respectively `jump_γ1` (l.11 ibid), `jump_γ2` (l.12 ibid)).

More importantly, the `Me` type parameter to `P'` represents type `JP.P` that extends `P'`. Type `Me` also captures the actual parameters *included* to `JP.P`, which makes it possible to write the return type of `γ1()` and `γ2()` more succinctly. Let, e.g., τ = `P<γ1, Me, Me, J_γ2>` be the return type of `γ1()`. The first two parameters to τ say that pushing `γ1`, results in a compound jump-stack, whose top element is `γ1`, and where the rest of the jump-stack is the current type. The third parameter to τ says that since `γ1` was pushed the result of a $\mathsf{jump}(\gamma_1)$ is the type of the receiver. The fourth parameter is `J_γ2` since a push of γ_1 does not change the result of $\mathsf{jump}(\gamma_2)$.

Some use cases for the encoded jump-stack data structure are in Figure 13. The type of variable `_1` encodes a stack into which `γ2`, `γ1`, `γ1` were pushed (in this order). Examining the type of `_2` we see that executing `jump_γ2` on `_1`, yields the empty stack in a single step. The type of `_3` is that state of the same stack after executing `jump_γ1`; it is exactly the same as popping a single element from the stack.

7 Proof of Theorem 1

On a first sight, the proof of Theorem 1 could follow the techniques sketched in Section 5 to type encode a DPDA (Definition 1). The partial transition function δ may be type encoded as in Figure 5(a), and the stack data structure of a DPDA can be encoded as in Figure 8.

The techniques however fail with ϵ-transitions, which allow the automaton to move between an unbounded number of configurations and maneuver the stack in a non-trivial manner, without making any progress on the input. The fault in the scheme lies with compile time computation being carried out by the $\mathsf{java}(\sigma)()$ functions, each converting their receiver type to the type of the receiver of the next call in the chain. We are not aware of a JAVA type encoding which makes it possible to convert an input type into an output type, where the output is computed from the input by an unbounded number of steps. [16]

The literature speaks of finite-delay DPDAs, in which the number of consecutive ϵ-transitions is uniformly bounded and even of realtime DPDAs in which this bound is 0, i.e., no ϵ-transitions. Our proof relies on a special kind of realtime automata, described by Courcelle [14].

▶ Definition 2 (Simple-Jump Single-State Realtime Deterministic Pushdown Automaton). *A simple-jump, single-state, realtime deterministic pushdown automaton (jDPDA, for short) is a triplet* $\langle \gamma, \gamma_1, \delta \rangle$ *where* γ *is a set of stack elements,* $\gamma_1 \in \gamma$ *is the initial stack element, and* δ *is the* partial transition function, $\delta : \gamma \times \Sigma \nrightarrow \gamma^* \cup j(\gamma)$,

$$j(\gamma) = \{\textit{instruction } \mathsf{jump}(\gamma) \mid \gamma \in \gamma\}.$$

A configuration of a jDPDA is some $c \in \gamma^*$ *representing the stack contents. Initially, the stack holds* γ_1 *only. For technical reasons, assume that the input terminates with* $\$ \notin \Sigma$*, a special end-of-file character.*

- *At each step a jDPDA examines* γ*, the element at the top of the stack, and* $\sigma \in \Sigma$*, the next input character, and executes the following:*
  1. *consume* σ
  2. *if* $\delta(\gamma, \sigma) = \zeta$, $\zeta \in \gamma^*$*, the automaton pops* γ*, and pushes* ζ *into the stack.*
  3. *if* $\delta(\gamma, \sigma) = \mathit{jump}(\gamma')$, $\gamma' \in \gamma$*, then the automaton repetitively pops stack elements up-to and including the first occurrence of* γ'*.*
- *If the next character is* \$*, the automaton may reject or accept (but nothing else), depending on the value of* γ*.*

In addition, the automaton rejects if $\delta(\gamma, \sigma) = \bot$ *(i.e., undefined), or if it encounters an empty stack (either at the beginning of a step or on course of a jump operation).*

As it turns out, every DCFG language is recognized by some jDPDA, and conversely, every language accepted by a jDPDA is a DCFG language [14]. The proof of Theorem 1 is therefore reduced to type-encoding of a given jDPDA. Towards this end, we employ the type-encoding techniques developed above, and, in particular, the jump-stack data structure (Figure 11).

[16] With the presumption that the JAVA compiler halts for all inputs (a presumption that does not hold for e.g., C++, and was never proved for JAVA), the claim that there is no JAVA type encoding for all DPDAs can be proved: Employing ϵ-transitions, it is easy to construct an automaton A^∞ that never halts on any input. A type encoding of A^∞ creates programs that send the compiler in an infinite loop.

Table 1 The transition function of a jDPDA A, $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$, $\gamma = \{\gamma_1, \gamma_2\}$ where γ_1 is the initial element.

	γ_1	γ_2
σ_1	$\mathsf{push}(\gamma_1, \gamma_1, \gamma_2)$	$\mathsf{push}(\gamma_2, \gamma_2)$
σ_2	$\perp$	$\mathsf{push}(\epsilon)$
σ_3	$\perp$	$\mathsf{jump}(\gamma_1)$
\$	accept	reject

Henceforth, let $k = |\gamma|$, $\ell = |\Sigma|$. The simple $k = 2$, $\ell = 3$ jDPDA A defined in Table 1 will serve as our running example. Let L be the language recognized by A.[17]

7.1 Main Types

Generation of a type encoding for a jDPDA starts with two empty types for sets L, Σ^*, where L represents the languages accepted by the jDPDA and Σ^* represents all words:

```
private static class ΣΣ    // Encodes set Σ*, type of reject
  { /* empty */  }
static class L extends ΣΣ // Encodes set L ⊆ Σ*, type of accept
  { /* empty */  }
```

(The full type encoding is in Figure 15 below; to streamline the reading, we bring excerpts as necessary.)

A configuration is encoded by a generic type `C`. Essentially, `C` is a representation of the stack, but $k+1$ type parameters are required:

- `Rest`, a type encoding of the stack after a pop (or jump with the top element), and,
- k types, named `JR`γ`1`, ... ,`JR`γk, encoding the type of `Rest` after $\mathsf{jump}(\gamma_1)$,... $\mathsf{jump}(\gamma_k)$.

Note that these $k+1$ parameters are sufficient for describing a configuration, i.e., if the top is γ_j, then for all $j \neq i$

$$\mathsf{jump}(\gamma_j) = \texttt{Rest.}\mathsf{jump}(\gamma_j)$$

In the special case of $\mathsf{jump}(\gamma_i)$ the returned type is still `Rest`, this is due to the fact that before a jump operation, we do not pop an element from the stack.

All instantiations of `C` must make sure that actual parameters are properly constrained, to ensure that they are (the type version of) pointers into the actual stack, not a trivial task, as will be seen shortly.

[17] Incidentally,

$$L = \left\{ w^* \mid w = (\sigma_1^n \sigma_2^m \sigma_3 | \sigma_1^n \sigma_2^n) \ , \ n > m, n > 1 \right\}$$

which is clearly not-regular; the equivalent BNF for L is:

$$S \to WS \,|\, \epsilon \ ; \ W \to D \,|\, AD\sigma_3 \ ; \ D \to \sigma_1 D \sigma_2 \,|\, \sigma_1\sigma_2 \ ; \ A \to A\sigma_1 \,|\, \sigma_1$$

Neither the BNF nor the representation are material for the proof.

```
static void isL(L l) {/**/}
static void accepts() {
  isL(A.build.$());
  isL(A.build.σ1().σ3().$());
  isL(A.build.σ1().σ2().$());
  isL(A.build.σ1().σ1().σ2().σ3().σ1().σ2().$());
}
static void rejects() {
  isL(A.build.σ1().$());
  isL(A.build.σ2().σ1().$());
  isL(A.build.σ1().σ2().σ3().$());
  isL(A.build.σ1().σ1().σ2().σ3().σ1().$());
}
```

Figure 14 Accepting and non-accepting call chains with the type encoding of jDPDA A (as defined in Table 1). All lines in `accepts()` type-check, while all lines in `rejects()` do not type-check.

In the running example, `C` is defined as:

```
interface C<       // Generic parameters:
  Rest extends C, // The rest of the stack, for pop or jump(γ) operations
  JRγ1 extends C, // Type of  Rest.jump(γ1), may be rest, or anything in it.
  JRγ2 extends C  // Type of  Rest.jump(γ2), may be rest, or anything in it.
>
{
  ΣΣ $();         // δ transition on end of input; invalid language by default
  C σ1();         // δ transition on σ1; dead end by default
  C σ2();         // δ transition on σ2; dead end by default
  C σ3();         // δ transition on σ3; dead end by default
  public interface E extends C<¤,¤,¤> { /* Empty stack configuration */ }
  interface ¤ extends C<¤,¤,¤> { /* Error configuration. */ }
}
```

This excerpt shows also classes `E` and `¤` which encode (as in Figure 11) the empty and the error configurations.

Type `C` defines $\ell + 1$ functions (4 in the example), one for each possible input character, and one for the end-of-file character defined as \$. Since `C` encodes an abstract configuration, return types of functions in it are the appropriate defaults which intentionally fail to emulate the automaton's execution. The return type of `$()` is $\Sigma\Sigma$ (rejection); the transition functions `σ1()`, ... `σℓ()`, return the raw type `C`.

7.2 Top-of-Stack Types

Types `Cγ1`, ... ,`Cγk`, specializing `C`, encode stacks whose top element is γ_1, ... ,γ_k. In A there are two of these:

```
interface Cγ1< // Configuration when γ1 is at top
  Rest extends C, JRγ1 extends C, JRγ2 extends C
> extends
  C<Rest, JRγ1, JRγ2>
{
}
interface Cγ2< // Configuration when γ2 is at top
  Rest extends C, JRγ1 extends C, JRγ2 extends C
> extends
  C<Rest, JRγ1, JRγ2>
{
}
static Cγ1<E,¤,¤> build = null;
```

In A, types `Cγ1` and `Cγ2` take three parameters; in general "Top of Stack" types take the aforementioned $k + 1$ parameters.

The method signatures of these types are generated using the mentioned parameters. The generating of methods will be discussed next.

The code defines the `static` variable `build`, the starting point of all fluent API call chains, to be of type `C`γ`1<E,¤,¤>`, i.e., the starting configuration of the automaton is a stack whose top is γ_1, and its `Rest` parameter is empty (`E`). Any of the two jumps possible on this rest results with, `¤`, an undefined stack. Examples of accepting and rejecting call chains starting at `A.build` can be seen in Figure 14.

7.3 Transitions

It remains to show the type encoding of δ, the transition function. Overall, there are a total of $k \cdot (\ell + 1)$ entries in a transition table such as Table 1. Conceptually, these are encoded by selecting the correct return type of functions σ`1()`, ... ,σk`()` and `$()` in each of the k "Top of Stack" types. Thanks to inheritance, we need to do so only in the cases that this return type is different from the default.

Overall, there are six kinds of entries in a transition table:

reject The default return type of `$()` in `C` is $\Sigma\Sigma$, which is *not* a subtype of `L`. Normally the result of a call chain that ends with `$()` cannot be assigned to a variable of type `L`. Moreover, since $\Sigma\Sigma$ is `private`, there is little that clients can do with this result.

accept The only case in which fluent call chain ending with `$()` can return type `L` is when the type returned of the call just prior to `.$()` covariantly changes the return type of `$()` to `L`.[18]

Recall that a jDPDA can only accept after its input is exhausted. In Table 1 we see that accept occurs when the top of the stack is γ_1. We therefore add to the body of type `C`γ`1` the line

```
@Override L $();
```

$\perp$ When a prefix of the input is sufficient to conclude it must be rejected however it continues, the transition function returns $\perp$. In A this occurs when the top of the stack is γ_1 and one of σ_2 or σ_3 is read. To type encode $\delta(\gamma_1, \sigma_2) = \perp$, one must *not* override σ`2()` in type `C`γ`1`; the inherited return type (l.15 Figure 15) is the raw `C`. Subsequent calls in the chain will all receive and return a raw `C` (Recall that all σi`()`, $i = 1, \ldots, \ell$, are functions in `C` that return a raw `C`). Therefore, the final `$()` will reject.

Two other situations in which a jDPDA rejects but not demonstrated in A are: a jump that encounters an empty stack, and reading a character from when the stack is empty. In our type encoding these are handled by the special types `E` and `¤` (ll.17–18 ibid), both extend `C` without overriding any of its methods. Again, remaining part of the call chain will stick to raw `C`s up until the final `$()` call rejects the input.

jump(γ_i) The design of the generic parameters makes the implementation of jump(γ_i) operations particularly simple. All that is required is to covariantly change the return type of the appropriate σj`()` function to the appropriate `JR`γi or `Rest` parameter (recall that a jump occurs after popping the current element from the stack, so we refer to `JR` type parameters rather than `J`'s).

In Table 1 we find that $\delta(\gamma_2, \sigma_3) =$ jump(γ_1). Accordingly, the type of σ`2()` in `C`γ`2` (l.34) is `JR`γ`1`.

[18] This is not to be confused with dynamic binding; types of fluent API call chains are determined statically.

push(ζ) Push operations are the most complex, since they involve a pop of the top stack element, and pushing any number, including zero, of new elements. The challenge is in constructing the correct $k+1$-parameter instantiation of `C`, from the current parameters of the type. Each of these $k+1$ is also an instantiation of `C` which may require more such parameters. Even though the number of ingredients is small, the resulting type expressions tend to be excessively long and unreadable.

The predicament is ameliorated a bit by the idea, demonstrated above with auxiliary type `P'` (Figure 12), of delegating the task of creating a complex type to an auxiliary generic type. The task of this sidekick is simplified if some of its generic parameters are sub-expressions that recur in the desired result.

Cases in point are $\delta(\gamma_1,\sigma_1) = \mathsf{push}(\gamma_1,\gamma_1,\gamma_2)$, and $\delta(\gamma_2,\sigma_1) = \mathsf{push}(\gamma_2,\gamma_2)$ of Table 1. The corresponding sidekick types, (γ`1`σ`1_Push_`γ`1`γ`1`γ`2` and γ`2`σ`1_Push_`γ`2`γ`2`) can be found in lines 36–43 of Figure 15. The first of these define the correct return type of σ`1()` in case γ`1` is the top element, the second of σ`2`, in case γ`2` is the top element. Examine now the definition of types `C`γ`1`,`C`γ`2` in the figure, and in particular lines 21–23 and 29–31 which define the list of types they extend. Notice that each extends one of the sidekicks, inheriting the covariant overrides of σ`1()`.

More generally, economy of expression may require that for each case of $\delta(\gamma,\sigma) = \mathsf{push}(\zeta)$ in the transition table, one creates a sidekick type which overrides the appropriate σ`()` function. The appropriate `C`γ type then inherits the definition from the sidekick.

Conclusion

The proof of Theorem 1 is an algorithm, taking as input some jDPDA, and returning as output a set of Java type definitions. The returned types, allow a call chain $\mathsf{java}(\alpha)$, such that the type of the returned object represents the configuration of the input automaton after reading α. If the automaton rejects after α, then the returned type is the illegal $\Sigma\Sigma$, and if the automaton accpets, the type shall be L.

8 The Prefix Theorem

▶ **Theorem 2.** *Let A be a DPDA recognizing a language $L \subseteq \Sigma^*$. Then, there exists a* Java *type definition, J_A for types* `L`, `A`, `C` *and other types such that the* Java *command*

```
C c = A.build.java(α);
```
(3)

type checks against J_A if an only if there exists $\beta \in \Sigma^*$ such that $\alpha\beta \in L$ and type `C` *is the configuration of A after reading α. Furthermore, for any such β, Theorem 1 applies such that the* Java *command*

```
L ℓ = A.build.java(αβ).$();
```
(4)

always type-checks. Finally, the program J_A can be effectively generated from A.

Informally, a call chain type-checks if and only if it is a prefix of some legal sequence. Alternatively, a call chain won't type-check if there is no continuation that leads to a legal string in L.

The proof resembles that of Theorem 1. We provide a similar implementation for a jump-stack [19], that will not compile under illegal prefixes.

[19] recall that the two formal constructs have the same expressive power

```
class A { // Encode automaton A
  private static class ΣΣ    // Encodes set Σ*, type of reject
    { /* empty */  }
  static class L extends ΣΣ // Encodes set L ⊆ Σ*, type of accept
    { /* empty */  }
  // Configuration of the automaton
  interface C<       // Generic parameters:
    Rest extends C, // The rest of the stack, for pop or jump(γ) operations
    JRγ1 extends C, // Type of  Rest.jump(γ1), may be rest, or anything in it.
    JRγ2 extends C  // Type of  Rest.jump(γ2), may be rest, or anything in it.
  >
  {
    ΣΣ $();          // δ transition on end of input; invalid language by default
    C σ1();          // δ transition on σ1; dead end by default
    C σ2();          // δ transition on σ2; dead end by default
    C σ3();          // δ transition on σ3; dead end by default
    public interface E extends C<¤,¤,¤> { /* Empty stack configuration */ }
    interface ¤ extends C<¤,¤,¤> { /* Error configuration. */ }
    interface Cγ1< // Configuration when γ1 is at top
      Rest extends C, JRγ1 extends C, JRγ2 extends C
    > extends
      C<Rest, JRγ1, JRγ2>
      ,γ1σ1_Push_γ1γ1γ2<Rest,JRγ1,JRγ2,Cγ1<Rest, JRγ1, JRγ2>>
    {
      @Override L $();
    }
    interface Cγ2< // Configuration when γ2 is at top
      Rest extends C, JRγ1 extends C, JRγ2 extends C
    > extends
      C<Rest, JRγ1, JRγ2>
      ,γ2σ1_Push_γ2γ2<Rest,JRγ1,JRγ2>
    {
      @Override Rest σ2();
      @Override JRγ1 σ3();
    }
    interface γ1σ1_Push_γ1γ1γ2<Rest extends C,JRγ1 extends C,JRγ2 extends C,
                               P extends Cγ1<Rest, JRγ1, JRγ2 >>{
      // Sidekick of δ(γ1, σ1) = push(γ1, γ1, γ2)
      Cγ2<Cγ1<P, Rest, JRγ2>,P,JRγ2> σ1();
    }
    interface γ2σ1_Push_γ2γ2<Rest extends C,JRγ1 extends C,JRγ2 extends C>{
      // Sidekick of δ(γ2, σ1) = push(γ2, γ2)
      Cγ2<Cγ2<Rest, JRγ1, JRγ2>, JRγ1, Rest> σ1();
    }
  }
  static Cγ1<E,¤,¤> build = null;
}
```

Figure 15 Type encoding of jDPDA A (as defined in Table 1).

The main difference between the two theorems is: in Theorem 1 we allowed illegal call chains to compile, but not return the required `L` type, while in Theorem 2 the illegal chain won't compile at all.

Since the code suggested by the proof is similar to the code presented above, only the differences will be discussed.

We will use the same running example, defined by Table 1.

8.1 Main Types

The main types here are a subset of the previously defined main types.

```
static class L // Encodes set L ⊆ Σ*, type of accept
  { /* empty */  }
public interface E { /* Empty stack configuration */ }
interface ¤ { /* Error configuration. */ }
```

First, type ΣΣ is removed. A call chain that doesn't represent a valid prefix won't compile, thus, there is no need for an error return type such as ΣΣ. Second, `interface C` is

```
static void accepts() {
  A.build.$();
  A.build.σ1().σ3().$();
  A.build.σ1().σ2().$();
  A.build.σ1().σ1().σ2().σ3().σ1().σ2().$();
}
static void rejects() {
  A.build.σ1().$();
  A.build.σ2();
  A.build.σ1().σ2().σ3();
  A.build.σ1().σ1().σ2().σ3().σ1().$();
}
```

Figure 16 Accepting and non-accepting call chains with the type encoding of jDPDA A (as defined in Table 1). All lines in `accepts` type-check, and all lines in `rejects` cause type errors.

removed. Without it, the configuration types won't have the methods σ`1()`, ... ,σk`()` and `$()` from the supertype. These inherited methods, is what differentiates the previous proof from the current. Classes `¤` and `E` are defined similarly, except now they don't extend any type.

8.2 Top-of-Stack Types

Types `C`γ`1`, ... ,`C`γk, still represent stacks with γ`1`, ... ,γk as their top element, this time, the methods are defined ad-hock, in each type (they are not added in this figure as they are added with the use of sidekicks). In A there are two such types:

```
interface Cγ1< // Configuration when γ1 is at top
  Rest, JRγ1, JRγ2
> extends
  γ1σ1_Push_γ1γ1γ2<Rest,JRγ1,JRγ2,Cγ1<Rest, JRγ1, JRγ2>>
{
}
interface Cγ2< // Configuration when γ2 is at top
  Rest, JRγ1, JRγ2
> extends
  γ2σ1_Push_γ2γ2<Rest,JRγ1,JRγ2>
{
}
static Cγ1<E,¤,¤> build = null;
```

Note, that the type parameters of the former types hasn't changed, since the model we are trying to implement, hasn't changed. These $k+1$ parameters still suffice for our cause.

In Figure 16, call chains in the `accepts()` method correctly type-checks (i.e., in L), while the chains in `rejects()` do not type-check (i.e., these prefixes have no continuation that can lead to a legal word in L), where the last method invocation generates an

"method ... is undefined for the type ... "

error message.

The main difference between Figure 16 and Figure 14 is that there is no need to use an auxiliary function `isL()` as in Figure 16 since now illegal prefixes do not type-check.

8.3 Transitions

Due to the changes we expressed, the transition table is encoded slightly different.

Encoding of the legal operations accept, jump(γ_i) and push(ζ) remains as in Theorem 1, since we want the same behavior for legal call chains. The minor differences are in the illegal operations reject and $\perp$:

```
1 static class A { // Encode automaton A
2   static class L // Encodes set L ⊆ Σ*, type of accept
3     { /* empty */  }
4   public interface E { /* Empty stack configuration */ }
5   interface ¤ { /* Error configuration. */ }
6   // Configuration of the automaton
7   interface Cγ1< // Configuration when γ1 is at top
8     Rest, JRγ1, JRγ2
9   > extends
10    γ1σ1_Push_γ1γ1γ2<Rest,JRγ1,JRγ2,Cγ1<Rest, JRγ1, JRγ2>>
11  {
12    L $();
13  }
14  interface Cγ2< // Configuration when γ2 is at top
15    Rest, JRγ1, JRγ2
16  > extends
17    γ2σ1_Push_γ2γ2<Rest,JRγ1,JRγ2>
18  {
19    Rest σ2();
20    JRγ1 σ3();
21  }
22  interface γ1σ1_Push_γ1γ1γ2<Rest,JRγ1,JRγ2,P extends Cγ1<Rest, JRγ1, JRγ2 >>{
23    // Sidekick of δ(γ1, σ1) = push(γ1, γ1, γ2)
24    Cγ2<Cγ1<P, Rest, JRγ2>,P,JRγ2> σ1();
25  }
26  interface γ2σ1_Push_γ2γ2<Rest,JRγ1,JRγ2>{
27    // Sidekick of δ(γ2, σ1) = push(γ2, γ2)
28    Cγ2<Cγ2<Rest, JRγ1, JRγ2>, JRγ1, Rest> σ1();
29  }
30  static Cγ1<E,¤,¤> build = null;
31 }
```

Figure 17 Type encoding of jDPDA A (as defined in Table 1) that allow a partial call chain, if and only if, there exists a legal continuation, that leads to a word in L (the language of A).

reject Since we add the methods ad-hock to each type, the reject entry means that the corresponding type, *won't* have a `$()` method, i.e., type `Cγ2` doesn't have a method `$()`.

$\perp$ We encounter $\perp$ on the transition function when some input character σ is not allowed for the top of the stack element γ. In that case, the corresponding type `Cγ` *must not* have a method for σ, this way, invoking the methods will result in type error. In Table 1 a $\perp$ may occur when the top of the stack is γ_1 and the input character is σ_2, thus, no method `σ2` is introduced in type `Cγ1`.

The use of sidekicks is still allowed and recommended to improve readability of code.

Conclusion

In this section, a proof, similar to the one in Section 7 is provided. An algorithm was introduced, to not only emulate the running of some jDPDA A, but also to "halt it" in the earliest time possible, i.e., only if there is no legal call chain from this point to result in a legal word in the language of A.

9 Notes on Practical Applicability

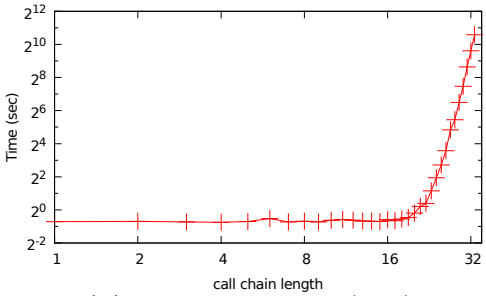
Theorem 1 and its proof above provide a concrete algorithm for converting an EBNF specification of a fluent API into its realization:

1. Convert the specification into a plain BNF form [20].

[20] `http://lampwww.epfl.ch/teaching/archive/compilation-ssc/2000/part4/parsing/node3.html`

```
static interface Cons<Car,Cdr>{
  Cons<
    Cons<Car,Cdr>,
    Cons<Car,Cdr>
  > d();
}
```

(a) Encoding of a binary type tree



(b) Compilation time (sec[a]) *vs.* length of call chain.

[a] measured on an Intel i5-2520M CPU @ 2.50GHz ×4, 3.7GB memory, Ubuntu 15.04 64-bit, `javac` 1.8.0_66

Figure 18 Exponential compilation time for a simple JAVA program.

2. Convert this BNF into a type of DPDA (using parsing algorithms e.g., LR(k), LALR, LL(k)). This conversion might fail [21].
3. Convert this DPDA into a jDPDA. (Conversion is guaranteed to succeed)
4. Apply the proof to generate appropriate JAVA type definitions, making sure to augment methods with code to maintain the fluent-call-list. Parsing the fluent-call-list can be done either in each method, or lazily, when the product of the fluent API call chain is to be used.

Although possible, a practical tool that uses the proof directly is a challenge. Part of the problem is the complexity of the algorithms used, some of which, e.g., the DPDA and jDPDA equivalence have never been implemented. Yet another issue that clients of compiler-compiler have grown to expect facilities such as means for resolving ambiguities, manipulation of attributes, etc. Also, for a fluent API to be elegant and useful, it should support method with parameters whose parameters are also defined by a fluent API: these two APIs may mutually recursive and even the same. Support of these features through four or so algorithmic abstractions may turn out to be a decent engineering task.

Yet another challenge is controlling the compiler's runtime. Learning that linear time parsers and lexical analyzers are possible, and being accustomed to seeing these in practice, one may expect the compiler would run in linear, or at least polynomial time. As it turns out, this time is exponential in the worst case (at least for `javac`). An encoding of a S-expression in type `Cons` (Figure 18(a)) is a not terribly complex such worst case.

Type `Cons` takes two type parameters, `Car` and `Cdr` (denoting left and right branches). Denote the return type of `d()` by

$$\tau = \texttt{Cons< Cons<Car, Cdr>, Cons<Car, Cdr> >}.$$

Let σ denote the type of the `this` implicit parameter to `d`. Now, since $\tau = \texttt{Cons<}\sigma,\sigma\texttt{>}$, we have $|\tau| \geq 2|\sigma|$, where the size of a type is measured, e.g., in number of characters in its textual representation. Therefore, in a chain of n calls to `d()`

$$\texttt{(Cons<?,?>(null)).}\overbrace{\texttt{d().}\cdots\texttt{.d()}}^{n\text{ times}}\texttt{;} \tag{5}$$

the size of the resulting type is $O(2^n)$.

[21] In the LR case, we know [30] there exists an equivalent grammar for which the conversion will succeed

Figure 18(b) shows, on the doubly logarithmic plane, the runtime (on a Lenovo X220) of the `javac` compiler (version 1.8.0_66) in face of a Java program assembled from Figure 18 and Equation 5 placed as the single command of `main()`. Exponential growth is demonstrated by the right-hand side of the plot, in which curve converges on a straight line. (In fact, a variation of the construction may lead to even super-exponential growth rate of the size of types.)

We believe that this exponential growth is due to a design flaw in the compiler. Had the compiler used a representation of types that allows sharing of expression types, compilation time would be linear.

Still, with current compiler technology, the type encoding scheme demonstrated in Figure 15 might not be scalable.

10 Conclusion and Future Work

The main contribution of this work is the proof that most useful grammars have a fluent API. This brings good news to library designers laboring at making their API slick, accessible, and more robust to mistakes of clients: If your API can be phrased in terms of a "decent" BNF, do not lose hope; the task may be Herculean, but it is (most likely) possible.

Other practitioners may appreciate the toolbox of type encodings offered here gaining better understanding of the computational expressiveness of Java generics and type hierarchy, and, a better tool for designing, experimenting with and perfecting fluent APIs.

However, once possibility was demonstrated theoretically, the next *research* challenge is in an actual fluent API compiler based on lighter weight parsing algorithms. Precisely, the challenge is in developing a parsing (or at least recognition) algorithm which is not only efficient but also falls within the limited computing power of the Java types.

On the theoretical front, one may ask whether our result is the best possible: Can the Java type system be coerced to recognize general (that is, nondeterministic) context-free languages?

As mentioned earlier, to the best of our knowledge, the complexity of the Java type checker has never been analyzed. In light of the empirical finding in Section 9, research in this direction may be worthwhile.

Other directions include formalizing the proof in Section 7 here and extensions to other languages.

On a philosophical perspective, several modern programming languages acquire high-level constructs at a staggering rate (C++ and Scala [35] being prominent examples). The main yardstick for evaluation these is "programmer's convenience". This work suggests an orthogonal perspective, namely computational expressiveness, or, stated differently, ranking of a new construct by its ability to recognize languages in the Chomsky hierarchy [12].

References

1 David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. C++ in Depth Series. Addison-Wesley, 2004.

2 Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

3 Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In Gary Leavens, editor, *Proc. of the 24th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'09)*, pages 1015–1022, Orlando, FL, USA, October 2009. ACM

Press. URL: http://doi.acm.org/10.1145/1639950.1640073, doi:10.1145/1639950.1640073.

4 Ken Arnold and James Gosling. *The* Java *Programming Language*. The Java Series. Addison-Wesley, Reading, MA, 1996.

5 Matthew H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley, 1998.

6 Jean-Michel Autebert, Jean Berstel, and Luc Boasson. *Context-Free Languages and Pushdown Automata*. Springer, 1997.

7 Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming. In *Advanced Functional Programming*, pages 28–115. Springer, 1999.

8 Nels E. Beckman, Duri Kim, and Jonathan Erik Aldrich. An empirical study of object protocols in the wild. In Mira Mezini, editor, *Proc. of the 25th Euro. Conf. on OO Prog. (ECOOP'11)*, volume 6813 of *LNCS*, pages 2–26, Lancaster, UK, June25-29 2011. Springer.

9 Kevin Bierhoff and Jonathan Erik Aldrich. Lightweight object specification with typestates. In Michel Wermelinger and Harald C. Gall, editors, *Proc. of the 10th European Soft. Eng. Conf. and 13th ACM SIGSOFT Symp. on the Foundations of Soft. Eng. (ESEC/FSE'05)*, pages 217–226, Lisbon, Portugal, September 2005. ACM Press.

10 Eric Bodden. TS4J : A fluent interface for defining and computing typestate analyses. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in* Java *Program Analysis - SOAP '14*, pages 1–6, 2014. URL: http://dl.acm.org/citation.cfm?doid=2614628.2614629http://www.bodden.de/pubs/bodden14ts4j.pdf, doi:10.1145/2614628.2614629.

11 Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Bjørn N. - Benson Freeman and Craig Chambers, editors, *Proc. of the 13th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'98)*, pages 183–200, Vancouver, BC, Canada, October18-22 1998. ACM SIGPLAN Notices 33(10).

12 Noam Chomsky. *Formal properties of grammars*. Addison-Wesley, 1963.

13 John Cocke. *Programming Languages and Their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York University, 1969.

14 Bruno Courcelle. On jump-deterministic pushdown automata. *Math. Sys. Theory*, 11:87–109, 1977.

15 James C. Dehnert and Alexander Stepanov. Fundamentals of generic programming. In *Generic Programming*, pages 1–11. Springer, 2000.

16 Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000. URL: http://portal.acm.org/citation.cfm?doid=352029.352035, doi:10.1145/352029.352035.

17 Charles Donnelly and Richard Stallman. *Bison*, 2015.

18 Jay Earley. An efficient context-free parsing algorithm. *Comm. of the ACM*, 13(2):94–102, 1970. doi:10.1145/362007.362035.

19 Sebastian Erdweg, Lennart C.L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Sugarj: Library-based language extensibility. In Kathleen Fisher, editor, *Proc. of the 26th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'10)*, pages 187–188, Portland OR, USA, October22-27 2011. ACM Press.

20 Steve Freeman and Nat Pryce. Evolving an embedded domain-specific language in Java. In Peri L. Tarr and William R. Cook, editors, *Proc. of the OOPSLA'06 Companion*. ACM Press, October22-26 2006.

21 Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In Ron Crocker and Guy L. Steele Jr., editors, *Proc. of the 18th Ann. Conf. on OO Prog. Sys., Lang., &*

Appl. (OOPSLA'03), pages 115–134, Anaheim, CA, USA, October 2003. ACM SIGPLAN Notices 38 (11). URL: http://www.informatik.uni-trier.de/~ley/db/conf/oopsla/oopsla2003p.html.

22 Joseph Gil and Zvi Gutterman. Compile time symbolic derivation with C++ templates. In *Proc. of the USENIX C++ Conf.*, pages 249–264, Santa Fe, NM, April 1998. USENIX Association.

23 Joseph (Yossi) Gil and Keren Lenz. Simple and safe SQL queries with templates. In Charles Consel, editor, *Proc. of the 6th Conf. on Generative Prog. & Component Eng.*, LNCS, pages 13–24, Salzburg, Austria, October 2007. ACM Press.

24 Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, MA, 1984.

25 Zvi Gutterman. Turing templates—on compile time power. Master's thesis, Technion—Israel Institute of Technology, 2003.

26 John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 2nd edition, 2001.

27 Claus Ibsen and Jonathan Anstey. *Camel in action*. Manning Publications Co., Shelter Island, NY, 2010.

28 JBoss Group. Hibernate product homepage. http://www.hibernate.org/, 2006.

29 Jevgeni Kabanov and Rein Raudjärv. Embedded typesafe domain specific languages for JAVA. *Proceedings of the 6th international symposium on Principles and practice of programming in* JAVA- *PPPJ '08*, 2008. doi:10.1145/1411732.1411758.

30 Donald Ervin Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965. doi:10.1016/S0019-9958(65)90426-2.

31 Robert Larsen. Fluenty: A type safe query API. Master's thesis, University of Oslo, 2012.

32 Peter Linz. *An Introduction to Formal Languages and Automata*. Jones & Bartlett Learning, 2011.

33 Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (ICMD'2006)*, Chicago, Illinois, 2006.

34 David R. Musser and Alexander A. Stepanov. Generic programming. In *Symbolic and Algebraic Computation*, pages 13–25. Springer, 1989.

35 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

36 Matthew M Papi. Practical pluggable types for JAVA. Master's thesis, Massachusetts Institute of Technology, 2008. URL: http://portal.acm.org/citation.cfm?doid=1390630.1390656, doi:10.1145/1390630.1390656.

37 Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 3rd edition, 1997.

38 David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2002.

39 Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

40 Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967. URL: http://www.sciencedirect.com/science/article/pii/S001999586780007X, doi:10.1016/S0019-9958(67)80007-X.

IceDust: Incremental and Eventual Computation of Derived Values in Persistent Object Graphs*

Daco C. Harkes[1], Danny M. Groenewegen[2], and Eelco Visser[3]

1 **Delft University of Technology**
`d.c.harkes@tudelft.nl`
2 **Delft University of Technology**
`d.m.groenewegen@tudelft.nl`
3 **Delft University of Technology**
`e.visser@tudelft.nl`

Abstract

Derived values are values calculated from base values. They can be expressed in object-oriented languages by means of getters calculating the derived value, and in relational or logic databases by means of (materialized) views. However, switching to a different calculation strategy (for example caching) in object-oriented programming requires invasive code changes, and the databases limit expressiveness by disallowing recursive aggregation.

In this paper, we present IceDust, a data modeling language for expressing derived attribute values without committing to a calculation strategy. IceDust provides three strategies for calculating derived values in persistent object graphs: Calculate-on-Read, Calculate-on-Write, and Calculate-Eventually. We have developed a path-based abstract interpretation that provides static dependency analysis to generate code for these strategies. Benchmarks show that different strategies perform better in different scenarios. In addition we have conducted a case study that suggests that derived value calculations of systems used in practice can be expressed in IceDust.

1998 ACM Subject Classification D.3.2 Data-flow languages

Keywords and phrases Incremental Computing, Data Modeling, Domain Specific Language

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.11

1 Introduction

Derived values are values calculated from base values (provided by users). When a base value changes, the derived values depending on it should change accordingly. Hence, the important events for interacting with derived values are writes to base values and reads of derived values. This specification of derived values leaves room for multiple strategies for calculating derived values. Derived values can be calculated when they are read or they can be cached and updated when the underlying base values change. The performance of these strategies depends on characteristics of the data model and usage scenarios. When neither of these calculation strategies provides acceptable performance, updates can be postponed, temporarily allowing reads to return outdated derived values.

Object-oriented programming languages express derived values through getters containing code that calculates a derived value, implying that the derived value is recalculated each time it is read. Switching to calculating the derived value when an underlying value changes,

* This research was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206).

or switching to eventually calculating the derived value, requires invasive code changes. By contrast, most relational databases allow easy switching between calculate-on-read and calculate-on-write as they support both materialized and non-materialized views for calculating derived values. However, relational databases only provide limited expressiveness for recursion, and do not support eventual calculation of derived values. Datalog provides more expressiveness than relational database views, but also limits recursion, and does not support eventual calculation of derived values.

This paper presents IceDust, a language supporting definition of attributes with derived values without committing to a calculation strategy. The compiler provides three different implementation strategies for calculating derived values: (1) Calculate-on-Read, which calculates the derived value every time it is read, (2) Calculate-on-Change, which maintains a cache incrementally by calculating the derived value every time an underlying value is changed, and (3) Calculate-Eventually, which schedules calculations of derived values, and thus sacrifices consistency temporarily. All these strategies allow unrestricted recursion, but do not provide termination guarantees. In particular, our contributions are:

- The IceDust language for data modeling with derived values (Section 2)
- A formal analysis of the dependencies in IceDust programs (Section 3)
- Three calculation strategies to satisfy different non-functional requirements (Section 4)
- Benchmarks showing the performance differences between the strategies (Section 5)
- A case study of migrating a custom eventual calculation system to IceDust (Section 6)

2 Declarative Data Modeling with Derived Values

This section discusses three issues of data modeling with derived values in object-oriented programming languages and show how data modeling in IceDust addresses these issues and leads to concise specifications. As running example we use (an aspect of) a learning management system in which students solve assignments. Figure 1 shows a Java implementation with classes `Assignment` and `Question`, where assignment represents a collection of questions and its progress is the average of the progress on the individual questions.

Bidirectional Relations. Object-oriented languages model bidirectional relations as properties in classes on both sides of the relation. Keeping these properties consistent requires code that has to be repeated for every bidirectional relation. Figure 1 includes five methods concerned with keeping the relation `Assignment-Question` consistent on updates: `setAssignment`, `addQuestion`, `removeQuestion`, `_addQ`, and `_remQ`. This pattern is identical for all one-to-many relations, but cannot be abstracted over in an object-oriented language. To avoid such boilerplate code, IceDust supports *bidirectional relations* as a language feature:

```
entity Assignment { }
entity Question { }
relation Assignment.questions * <-> 1 Question.assignment
```

These bidirectional relations are named on both sides of the relation, inspired by Object-Role Modeling [17]. The IceDust compiler keeps both sides of the association consistent without additional boilerplate code.

Native Multiplicities. Explicit collections and possible null values in object-oriented languages lead to boilerplate code to deal with the cardinalities of values returned by an expression. Operators in object-oriented languages are defined for operands with a cardinality of exactly one. Safely applying an operator to a nullable operand, requires a `null`-check.

```
public class Assignment {
  public Double getAverageProgress() {  return calculateAverageProgress();  }
  public Double calculateAverageProgress() {
    Stream<Double> progresss =
      questions.stream().map(q -> q.getProgress()).filter(p -> p != null);
    OptionalDouble average = progresss.mapToDouble(p -> p).average();
    return average.isPresent() ? average.getAsDouble() : null;
  }
  private Collection<Question> questions;
  public Collection<Question> getQuestions(){ return new HashSet<>(questions); }
  public void addQuestion(Question q) {  q.setAssignment(this);  }
  public void removeQuestion(Question q) {  q.setAssignment(null);  }
  protected void _addQ(Question q) {  questions.add(q);  }
  protected void _remQ(Question q) {  questions.remove(q);  }
}
public class Question {
  private Assignment assignment;
  public Assignment getAssignment() {  return assignment;  }
  public void setAssignment(Assignment a) {
    if(assignment != null) { assignment._remQ(this); }
    if(a != null) { a._addQ(this); }
    assignment = a;
  }
  private Double progress;
  public Double getProgress() {  return progress;  }
  public void setProgress(Double p) {  progress = p;  }
}
```

Figure 1 Object-oriented assignment system (implementation strategy: Calculate-on-Read).

Applying an operator to a collection of values, requires lifting it to a map. For example, accessing the progress of each individual question in Figure 1 is encoded as

```
questions.stream().map(q -> q.getProgress()).filter(p -> p != null)
```

IceDust adopts *native multiplicities* [18], delegating the handling of the cardinality of values returned by an expression to the language. For example, retrieving the progress for all questions is simply a projection:

```
questions.progress
```

Language constructs to get expressions of cardinality exactly one, such as `map`, `filter`, and `!= null`, are no longer required, as the type system knows how many values an expression returns (multiplicity denoted by `~`, where `*` is [0,n), `+` is [1,n), `?` is [0,1], and `1` is [1,1]):

```
mathAssignment                          // : Assignment ~ 1
mathAssignment.questions                // : Question   ~ *
mathAssignment.questions.progress       // : Float      ~ *
avg(mathAssignment.questions.progress) // : Float      ~ ?
```

Sometimes it is still necessary to reflect explicitly on the cardinality of a value. To that end one can use the `count` operator, for example, for counting the number of questions:

```
count(questions)
```

Reflection on the cardinality of values is also often used to select an alternative if no value is present. For specifying alternatives the choice operator (`<+`) can be used:

```
input <+ myDefault //if (count(input) > 0) input else myDefault
```

```
//Take all Code from Calculate–on–Read and add/change the following:
public class Assignment {
 private Double cachedAvgProgress;
 public Double getAverageProgress() {  return cachedAvgProgress;  }
 public void updateAvgProgress(){cachedAvgProgress = calculateAverageProgress();}
 protected void _addQ(Question q){ questions.add(q);    updateAvgProgress(); }
 protected void _remQ(Question q){ questions.remove(q); updateAvgProgress(); }
}
public class Question {
 public void setProgress(Double p){progress = p; assignment.updateAvgProgress();}
}
```

Figure 2 Object-oriented assignment system (implementation strategy: Calculate-on-Write).

Derived Value Attributes. Last but not least, object-oriented languages force early decisions on the implementation strategy for calculating derived values. In an object-oriented language, a derived value calculation can be expressed with a method that computes the value. However, this encodes a Calculate-on-Read implementation strategy. For cheap calculations or calculations that are done infrequently that may be fine. But for others, it may be necessary to cache the calculated value. Such an alternative computation strategy requires an invasive redefinition of the implementation. For example, Figure 2 implements a caching strategy for the `getAverageProgress` computation of Figure 1. Instead of computing the average on read, it is computed on writes of `progress` and `questions`. For this example, the impact of the change was relatively minor because in Figure 1 we had already factored `calculateAverageProgress` into a separate method. However, in real code the impact is typically non-trivial. In particular, because the introduction of a cached value requires taking into account all of its dependencies in order to trigger recomputation on any change that affects it. For example, `averageProgress` depends on `progress` and `questions`. Thus, `setProgress`, `_addQ`, and `_remQ` need to trigger recalculation of `averageProgress`.

IceDust provides *derived value attributes* for declarative specification of the value of attributes in terms of other attributes without committing to an implementation strategy:

```
entity Assignment {  avgProgress : Float? = avg(question.progress)  }
```

This separation of concerns enables focusing on specification of the logic of the derived value. The derived value expression specifies what the value of the attribute should be. Derived value attributes in IceDust support recursive definitions, including recursive aggregation (which is not supported in materialized views or stratified Datalog):

```
entity Assignment {  progress : Float? = avg(children.progress)  }
relation Assignment.parent ? <–> * Assignment.children
```

Language Definition. We have combined the ideas for improving data modeling by means of bidirectional relations, native multiplicities, and derived value attributes in the design of the experimental IceDust language. In order to embed IceDust data models in full fledged web applications the compiler generates code in the WebDSL programming language [36].

The design of IceDust was heavily influenced by previous work on relations as a first-class language construct. From Rumer [4] and RelJ [5] we adopt the restriction to binary, bidirectional relations. From the Relations language [18] we adopt the syntax of declarations and property access, integrating multiplicities in relations. Multiplicities derive from the work of Steimann [32, 33], which extends an object-oriented language with multiplicity annotations to support uniform treatment of values of different cardinality and avoids the boilerplate code

```
Program ::= model Entity* Relation*
Entity ::= entity E { Attribute* }
Relation ::= relation E.r m <-> m E.r
Attribute ::= a : T m
            | a : T m = e
            | a : T m = e (default)
T ∈ PrimitiveType ::= Boolean | Int | Float | Datetime | String
m ∈ Multiplicity ::= ? | 1 | * | * (ordered) | + | + (ordered)
e ∈ Expr ::= f ( e ) | e1 ⊕ e2 | ! e | e1 ? e2 : e3 | e . a | e . r | e as T | this | Literal
Literal ::= true | false | null | int | float | datetime | string
f ∈ AggrOp ::= min | max | avg | sum | concat | count | conj | disj
⊕ ∈ BinOp ::= + | - | * | / | % | && | || | > | >= | < | <= | == | =! | <+ | ++
```

Figure 3 Syntax of the IceDust data modeling language.

required to support different multiplicities. We adopt the integration of such multiplicities in the type system (dubbed native multiplicities) of the Relations language [18].

Figure 3 defines the grammar of IceDust. `E`, `a` and `r` are entity, attribute and relation names respectively. An IceDust program consists of entities and relations. Entities contain three kinds of attributes: 'normal' attributes (`a : T m`), derived value attributes (`a : T m = e`), and default value attributes (`a : T m = e (default)`). Users can set the value of 'normal' attributes and read the value later. Users cannot set the value of derived value attributes, but they can read the value calculated with expression `e`. Finally, users can set the value of default value attributes and read the value later, but they can also set the value to null (or not set it all) and read the value calculated by `e`. Attributes are limited to primitive types, as an entity type would create a unidirectional relation (which would give problems in the dependency analysis). A relation defines a bidirectional relation with a name and multiplicity on both sides. The domain of the expression language is primitive types (Boolean, Int, Float, Datetime and String) and objects. The language covers object graph navigation and calculations over the primitive types. Note the aggregation operations over primitive types to deal with multiplicities `*` and `+`. The expression language is expressive enough to specify derived values, and simple enough to allow multiple implementation strategies.

The type system of IceDust is mostly concerned with native multiplicities. A type in IceDust is a tuple of two lattice values (Figure 4). The primitive types, the declared entities, the `null` and error type form a lattice. Multiplicity and ordering form another lattice. During derived value calculation all values are read-only in IceDust. A value which is lower or equal in both lattices can be used in a place where a certain type, multiplicity, and ordering is expected. For example, a `Float?` can be supplied where a `Float*` is expected.

3 Dependency and Data Flow Analysis

IceDust specifications define the value of attributes in terms of other attributes. These definitions are declarative in the sense that they abstract from the implementation strategy used to calculate the values. In the next section we define three implementation strategies for the calculation of attribute values: Calculate-on-Read, Calculate-on-Write, and Calculate-Eventually. The latter two strategies require dependency and data flow information. In this section we define the computation of dependencies between attributes by means of a path-based abstract interpretation of expressions. Since IceDust does not have statements, data

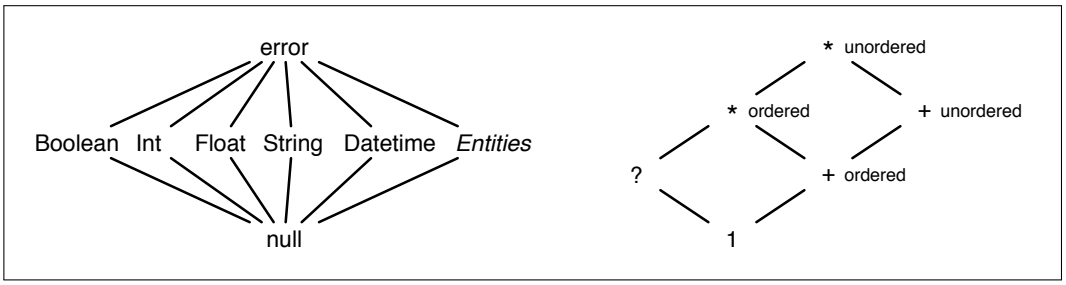


Figure 4 IceDust's type lattice (left), and multiplicity and ordering lattice (right).

flow coincides with control flow, and the data flow relation is the inverse of the dependency relation. The static dependency and data flow analysis is performed in three steps: (1) compute attribute dependencies by means of path-based abstract interpretation, (2) reverse the dependencies to construct the data flow relation, and (3) organize the data flow in a graph and extract strongly connected components with a topological ordering.

Example. To illustrate the analysis we use a more complex version of the learning management system (Figure 5). This example features a tree of assignments, and grade calculation logic for submissions by students to these assignments. `Assignments` are structured in a tree through the `parent-children` relation. A `Submission` represents the solution created for an assignment by a student. Leaf submissions are graded by assigning a grade to the `grade` attribute (overriding the default value), while the grades of non-leaf submissions depend (indirectly) on the grades of their child submissions:

```
grade      : Float?  = if(childPass) childGrade else null (default)
pass       : Boolean = grade >= (assignment.minimum<+0.0) && childPass <+ false
childGrade : Float?  = avg(children.grade)
childPass  : Boolean = conj(children.pass)
```

Note that students only receive a grade for a collection-submission if all of the child submissions are `pass`, and a submission is only a `pass` when its `grade` is above the `minimum` assignment grade and all its children pass. The `minimum` for an assignment is optional, without `minimum` the grade should be higher than or equal to 0.0, which is always true. Submissions are (one of) the `best` of an assignment when their `grade` equals the highest grade. Finally, every assignment has an average grade and pass percentage. This example is interesting for dependency analysis as it features mutually recursive definitions of `grade`, `pass`, `childGrade` and `childPass` through the `parent-child` relation of `Submission`.

Step 1: Dependencies. The dependencies of an attribute are all the attributes and relations that are needed to compute the derived value of that attribute. The dependencies are reachable from the entity of the attribute via a path. A dependency is denoted by $(Ent.Attr \leftarrow \pi)$, where $Ent.Attr$ is the attribute and π is the path to an attribute or relation.

Computing the dependencies requires extracting paths from expressions defining derived values. The *path-based abstract interpretation* relation (Figure 6) defines the dependency paths of an expression. We use the notation $Expr \searrow \{\pi\}\{\rho\}$, where $Expr$ is the expression that is abstractly interpreted, and $\{\pi\}$ and $\{\rho\}$ are the sets of paths defined by the abstract interpretation. The paths in $\{\pi\}$ are extensible, while the paths in $\{\rho\}$ are not. All paths start with `this` [This] or with object graph navigation [NavStart]. When navigating the object graph by means of `e.attrOrRel` all dependency paths $\{\pi\}$ in `e` are extended with

```
entity Assignment {
  name       : String
  question   : String
  minimum    : Float?
  avgGrade   : Float? = avg(submissions.grade)
  passPerc   : Float? = sum(submissions.passInt) / count(submissions) * 100.0
}
entity Student {
  name       : String
}
entity Submission {
  name       : String  = assignment.name + " " + student.name
  answer     : String?

  grade      : Float?  = if(childPass) childGrade else null (default)
  pass       : Boolean = grade >= (assignment.minimum<+0.0) && childPass <+ false
  childGrade : Float?  = avg(children.grade)
  childPass  : Boolean = conj(children.pass)

  passInt    : Int     = if(pass) 1 else 0
  best       : Boolean = grade == max(assignment.submissions.grade) <+ false
}
relation Assignment.parent     ? <-> * Assignment.children
relation Submission.parent     ? <-> * Submission.children
relation Submission.student    1 <-> * Student.submissions
relation Submission.assignment 1 <-> * Assignment.submissions
```

Figure 5 Example program for dependency analysis.

`attrOrRel` [Nav]. The `if` [If] only allows extension of paths in the second and third operand, so Π_1 is passed to $\{\rho\}$. Operators with multiple operands take the union of the paths of their operands [Op], unary operators pass on paths [Not,Cast,Aggr], and literals do not contain any paths at all [Literal]. Path-based abstract interpretation of the expression defining `pass`

```
grade >= (assignment.minimum <+ 0.0) && childPass <+ false
```

produces a set containing the following paths:

```
grade
assignment.minimum
childPass
```

The *dependencies* relation (Figure 6) defines the dependencies of an attribute, entity and program. We use the notation $Attr|Ent|Prog \searrow\searrow \{(Ent.Attr \leftarrow \pi)\}$, where $Attr|Ent|Prog$ is an attribute, entity or program, and $\{(Ent.Attr \leftarrow \pi)\}$ is a set of dependencies. When an attribute depends on a value at the end of a path, it also depends on the relations en route. So, the rule for attributes [Att] takes the transitive prefix of the paths of its expression. As paths are concatenated later, and a `this` keyword in the middle would produce an invalid path, the `this` is removed from paths. As an example, the dependencies of `pass` are:

```
(Submission.pass ← grade)
(Submission.pass ← assignment.minimum)
(Submission.pass ← assignment)
(Submission.pass ← childPass)
```

The dependencies in for the individual attributes together constitute the dependencies for a full program [Ent,Prog].

Path-based abstract interpretation $\boxed{Expr \searrow \{\pi\}\{\rho\}}$

$$\frac{}{\text{this} \searrow \{\text{this}\}\{\}}\ [\text{This}] \qquad \frac{e \in Literal}{e \searrow \{\}\{\}}\ [\text{Literal}]$$

$$\frac{}{attrOrRel \searrow \{attrOrRel\}\{\}}\ [\text{NavStart}] \qquad \frac{e \searrow \Pi\ \mathrm{P}}{!\ e \searrow \Pi\ \mathrm{P}}\ [\text{Not}]$$

$$\frac{e \searrow \Pi\ \mathrm{P}}{e\ .\ attrOrRel \searrow \{\pi\ .\ attrOrRel \mid \pi \in \Pi\}\ \mathrm{P}}\ [\text{Nav}] \qquad \frac{T \in PrimitiveType \quad e \searrow \Pi\ \mathrm{P}}{e \text{ as } T \searrow \Pi\ \mathrm{P}}\ [\text{Cast}]$$

$$\frac{\oplus \in BinOp \quad e_1 \searrow \Pi_1\ \mathrm{P}_1 \quad e_2 \searrow \Pi_2\ \mathrm{P}_2}{e_1 \oplus e_2 \searrow \Pi_1 \cup \Pi_2 \quad \mathrm{P}_1 \cup \mathrm{P}_2}\ [\text{Op}] \qquad \frac{f \in AggrOp \quad e \searrow \Pi\ \mathrm{P}}{f(e) \searrow \Pi\ \mathrm{P}}\ [\text{Aggr}]$$

$$\frac{e_1 \searrow \Pi_1\ \mathrm{P}_1 \quad e_2 \searrow \Pi_2\ \mathrm{P}_2 \quad e_3 \searrow \Pi_3\ \mathrm{P}_3}{e_1\,?\,e_2 : e_3 \searrow \Pi_2 \cup \Pi_3 \quad \Pi_1 \cup \mathrm{P}_1 \cup \mathrm{P}_2 \cup \mathrm{P}_3}\ [\text{If}]$$

Dependencies $\boxed{Attr|Ent|Prog \searrow\!\searrow \{(Ent.Attr \leftarrow \pi)\}}$

$$\frac{e \searrow \Pi\ \mathrm{P} \quad E = \text{entity-of}(attr) \quad \Pi_2 = \bigcup \{\text{trans-pref}(\text{remove-this}(\pi)) \mid \pi \in \Pi \cup \mathrm{P}\}}{attr : T\ m\ \{= e, = e\ (\text{default})\} \searrow\!\searrow \{(E.attr \leftarrow \pi) \mid \pi \in \Pi_2\}}\ [\text{Att}]$$

$$\frac{}{\text{entity } t\ \{a^*\} \searrow\!\searrow \bigcup \{dep \mid a \searrow\!\searrow dep,\ a \in a^*\}}\ [\text{Ent}]$$

$$\frac{}{\text{model } E^*\ R^* \searrow\!\searrow \bigcup \{dep \mid E \searrow\!\searrow dep,\ E \in E^*\}}\ [\text{Prog}]$$

$\text{remove-this}(\text{this}\ .\ \pi) = \pi$
$\text{remove-this}(attrOrRel\ .\ \pi) = attrOrRel\ .\ \pi$
$\text{trans-pref}(\pi\ .\ attrOrRel) = \{\pi\ .\ attrOrRel\} \cup \text{trans-pref}(\pi)$
$\text{trans-pref}(attrOrRel) = \{attrOrRel\}$

Figure 6 Dependency analysis step 1: path-based abstract interpretation

Dependency inversion $\boxed{(Ent.Attr \leftarrow \pi) \nearrow (Ent.AttrOrRel \rightarrow \pi)}$

$$\frac{E_2 = \text{entity-of}(attrOrRel)}{(E\ .\ attr \leftarrow \pi\ .\ attrOrRel) \nearrow (E_2\ .\ attrOrRel \rightarrow \text{inv-path}(\pi)\ .\ attr)}\ [\text{InvDep}]$$

$\text{inv-path}(\pi\ .\ attrOrRel) = attrOrRel^{-1}\ .\ \text{inv-path}(\pi)$
$\text{inv-path}(attrOrRel) = attrOrRel^{-1}$
$\text{inv-path}(\text{null}) = \text{null}$

Data flow $\boxed{Prog \nearrow\!\nearrow \{(Ent.AttrOrRel \rightarrow \pi)\}}$

$$\frac{\text{model } E^*\ R^* \searrow\!\searrow Dep}{\text{model } E^*\ R^* \nearrow\!\nearrow \{df \mid dep \nearrow df,\ dep \in Dep\}}\ [\text{Prog}]$$

Figure 7 Dependency analysis step 2: data flow

Data flow graph $\boxed{Prog \nearrow\!\nearrow\!\nearrow (\{AttrOrRel\}, \{(AttrOrRel, Attr)\})}$

$$\frac{\text{model } E^*\ R^* \nearrow\!\nearrow DFlow \quad E = \{(x,y) \mid (Ent.x \rightarrow \pi.y) \in DFlow\} \quad V = \{x \mid (x,y) \in E\} \cup \{y \mid (x,y) \in E\}}{\text{model } E^*\ R^* \nearrow\!\nearrow\!\nearrow (V, E)}\ [\text{Prog}]$$

Figure 8 Dependency analysis step 3: data flow graph.

Step 2: Data Flow. The data flow of an attribute or relation is the set of all the attributes that depend on it to compute their derived value. The data flow relation is the inverse of the dependency relation. We write $(Ent.AttrOrRel \rightarrow \pi)$ to denote the data flow relation from the source, $Ent.AttrOrRel$, to the target, the end of the path π.

The *dependency inversion* relation, $(Ent.Attr \leftarrow \pi) \nearrow (Ent.AttrOrRel \rightarrow \pi)$, in Figure 7 defines the inverse of a dependency. A dependency is inversed by swapping source and target, and inverting the path π to get the path from target to source. The function `inv-path(`π`)` inverts the names in on path, and inverts their order. Name inversion is selecting the name on the opposing side of a relation; all relations in IceDust are bidirectional, and have names on both sides. All names in π can be inverted because they are relations. (π is the prefix of a full path, and only the last name of a path can be an attribute.) If the dependencies of the attribute `pass` are inversed the resulting data flow is:

```
(Submission.grade      → pass)
(Assignment.minimum    → submissions.pass)
(Submission.assignment → pass)
(Submission.childPass  → pass)
```

Step 3: Data Flow Graph. The data flow graph relation $Prog \nearrow\!\!\!\nearrow (V, E)$ in Figure 8, defines a data flow graph in terms of the data flow relation. The nodes in the graph are the attributes and the relations in an IceDust program. The edges (x, y) in the graph are $(AttrOrRel, Attr)$ from the data flow relation $(Ent.AttrOrRel \rightarrow \pi.Attr)$. Using Tarjan's algorithm [35] we find strongly connected components and a topological ordering for the data flow. The strongly connected components correspond to recursive dependencies.

The data flow graph for our example application is shown in Figure 9. The attributes `grade`, `pass`, `childGrade`, and `childPass` mutually depend on each other, a cycle in the graph (group 6). (The data flow is not cyclic: data flows up the submission tree.) The `minimum` precedes group 6 in the topological ordering, as `pass` in group 6 depends on it but minimum itself depends on nothing. On the other hand, the `passPerc`, `averageGrade`, and `best` depend on the results in group 6. The derived `name` for submissions is disconnected from the grade calculation, as the name of the submission does not have anything to do with the grade calculation. Relations only flow to attributes, and not vice versa. In IceDust, relations cannot be derived. This limits the expressiveness of IceDust, but also avoids 'dynamic dependencies', dependencies that are discovered while computing derived values.

Topological ordering can be used to statically schedule the computation of derived values. This is used in stratified Datalog, where a topological sort of the dependencies between rules is used to determine the order of computation [3, 13]. We will elaborate on computation scheduling, and on similarities with existing approaches, in later sections.

4 Implementation Strategies

The declarative specification of derived values in IceDust allows deferring the decision about implementation strategy from implementation to compilation time, and allows switching strategies to realize different non-functional requirements without invasive code changes. In this section we present three implementation strategies: Calculate-on-Read, Calculate-on-Write, and Calculate-Eventually. For each of these we have a compilation scheme that specifies what code to generate for IceDust's concepts.

On a high level the difference between the generated code for the different implementation strategies is the point in time at which derived values are calculated. Figure 10 shows the

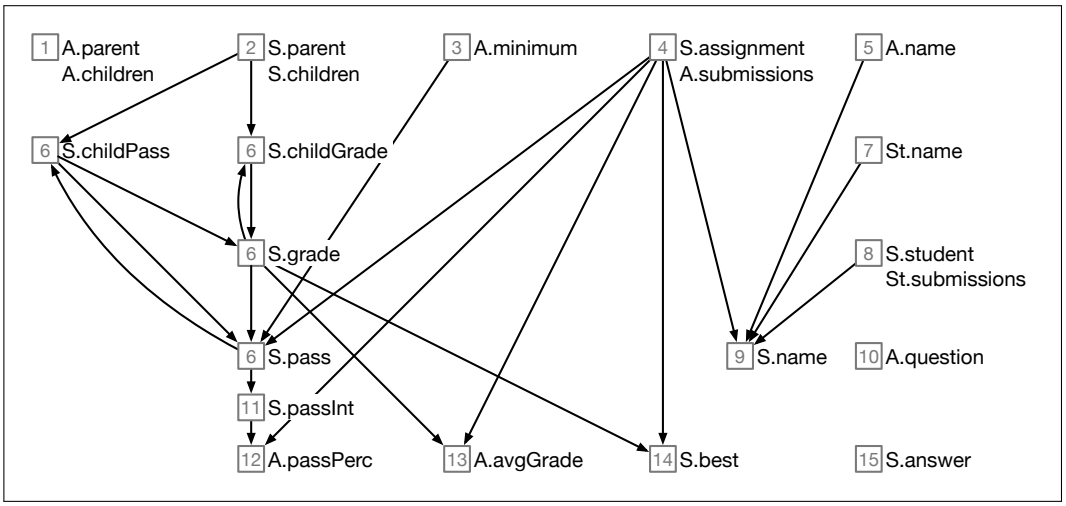


Figure 9 Step 3 example: strongly connected components and topological ordering in data flow.

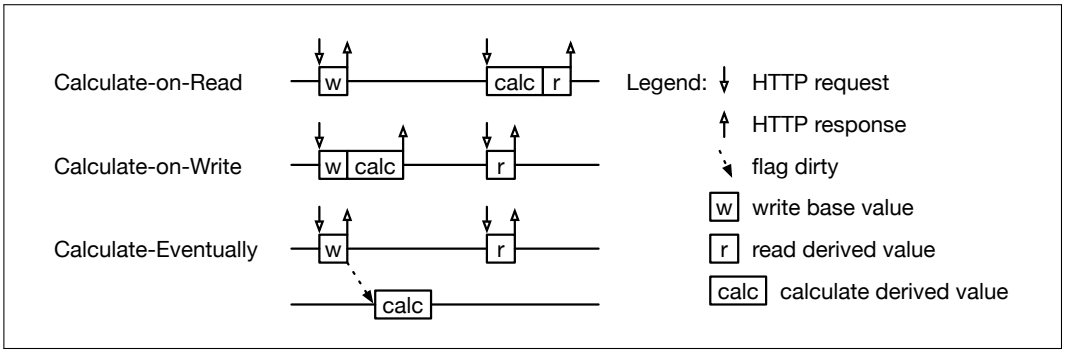


Figure 10 Thread activation diagrams for code generated by different implementation strategies.

differences by means of thread activation diagrams in response to incoming HTTP requests. The code generated by Calculate-on-Read calculates derived values when they are read. This means that writes to base values, on which derived values can depend, will be fast, but reads of derived values will be slow. The code generated by Calculate-on-Write calculates the derived values that depend on changed base values right away. Writes will be slow, but reads will be fast. The code generated by Calculate-Eventually schedules calculation of derived values on a separate thread. Writes and reads will be fast, but consistency is not guaranteed: possibly outdated derived values might be read.

Compiling to WebDSL IceDust is used to specify the data model and derived values for web applications. Our compiler compiles IceDust specifications to the WebDSL web programming language [36], which is a high-level target language for the implementation of data models. WebDSL persists its data in a relational database. This provides (1) data safety in case of a power outage, (2) enables large data sets, and (3) enables concurrent data access for concurrent HTTP requests. WebDSL's data modeling language is close to IceDust; it features entities and attributes (including Calculate-on-Read derived values):

```
entity Assignment {
  name     : String
  avgGrade : Float := avg([s.grade | s:Submission in subs where s.grade!=null])
}
```

Note the list comprehension syntax for applying a map to access the grade for each submission and filter on null values. WebDSL does not have bidirectional relations like IceDust, but it does have inverse properties:

```
entity Submission {  assignment  : Assignment (inverse = submissions)  }
entity Assignment {  submissions : Set<Submission>                      }
```

If a property is an inverse of another property, WebDSL keeps the values in the properties consistent. Our compiler targets these inverse properties for bidirectional relations, giving us the consistency of bidirectional relations for free.

With WebDSL already providing data persistence, large data sets, concurrency, Calculate-on-Read derived values, and inverse property consistency, our compilation schemes can focus on the essentials: default value behavior, multiplicities, bidirectional relations, and the Calculate-on-Write and Calculate-Eventually implementations for derived values.

The rest of this section describes the three implementation strategies in detail, using MorphJ[20]-style code generation templates for the compilation schemes. The templates use WebDSL (`black` with `purple keywords`) as target language and template-level control statements (*`blue italic`*) that iterate over entities, attributes, relations, and data flow edges (*`orange italic`*). We explain WebDSL code along the way, using callouts (for example: [1]) to refer to specific parts of generated WebDSL code.

Calculate on Read. Figure 11 defines the Calculate-on-Read compilation scheme. To translate IceDust with Calculate-on-Read to WebDSL we need to translate three IceDust features: (1) multiplicities, (2) default value attributes, and (3) bidirectional relations.

Multiplicities `?` and `1` are translated to WebDSL primitives, while multiplicities `*` or `+` are translated to lists. The getter for a normal attribute [2] (see Figure 11) is static for null-safety, it might be called on a null value, for example: `Assignment.get_passPerc(null)`. The getter is lifted to deal with a list of entities for which the attribute is referenced [10]. Attributes can only have multiplicity `?` or `1`, so there is no generation for multiplicity `*` or `+`. (List typed attributes would create overhead in WebDSL's mapping to the underlying database.)

Default value attributes are translated to two attributes [6,7] and one getter [9] in WebDSL. The first attribute [6] corresponds to the value possibly set by the user. The second attribute [7] corresponds to the default value expression. The getter [9] will return the user provided value, if any, and otherwise the default value. When only [6] is used to write values, and only [9] is used to read values the default value attribute will have IceDust's semantics. WebDSL features no `private` attributes and methods, so this behavior cannot be encapsulated.

Bidirectional relations are translated to properties and inverse properties (which are kept consistent by WebDSL). The right-hand side of the relation is translated to a normal WebDSL property [12,14,16], and the left-hand side is translated to a property with an inverse [11,13,15]. Unordered to-many relations are translated to sets, while the ordered relations are translated to lists. It would suffice to translate them all to lists, but WebDSL's relational database mapping has more overhead for lists than for sets. Relation navigation is overloaded on multiplicity: navigate from single entity via a to-one relation [17], or via a to-many relation [19], and navigate from multiple entities via a to-one relation [18], or via a to-many relation [20].

```
for E in Entities
  entity E {
    for a : T m in E.attributes
      a : T (default=null)[1]

      static function get_a(e : E) : T {  return if(e == null) null else e.a;  }[2]

    for a : T m = e1 in E.attributes
      a : T := calculate_a()[3]

      function calculate_a() : T {  return e1;  }[4]

      static function get_a(e : E) : T {  return if(e == null) null else e.a;  }[5]

    for a : T m = e1 (default) in E.attributes
      a : T (default=null)[6]
      a_default : T := calculate_a()[7]

      function calculate_a() : T {  return e1;  }[8]

      static function get_a(e : E) : T {
        return if(e == null) null else if(e.a == null) e.a_default else e.a;
      }[9]

    for a : T m {, = e1, = e1 (default)} in E.attributes
      static function get_a(entities : [E]) : [T] {
        return [E.get_a(e) | e : E in entities where E.get_a(e) != null];
      }[10]

    for relation E.l {1,?} <-> m2 E2.r in Relations
      l : E2 (inverse=r)[11]
    for relation E2.r m2 <-> {1,?} E.l in Relations
      l : E2[12]
    for relation E.l {*,+} (unordered) <-> m2 E2.r in Relations
      l : {E2} (inverse=r)[13]
    for relation E2.r m2 <-> {*,+} (unordered) E.l in Relations
      l : {E2}[14]
    for relation E.l {*,+} (ordered) <-> m2 E2.r in Relations
      l : [E2] (inverse=r)[15]
    for relation E2.r m2 <-> {*,+} (ordered) E2.r in Relations
      l : [E2][16]

    for relation E.l {1,?} <-> m2 E2.r
    and relation E2.r m2 <-> {1,?} E.l in Relations
      static function get_l(e : E) : E2 { return if(e == null) null else e.l; }[17]

      static function get_l(entities : [E]) : [E2]{
        return [E.get_l(e) | e : E in entities where E.get_l(e) != null];
      }[18]

    for relation E.l {+,*} <-> m2 E2.r
    and relation E2.r m2 <-> {+,*} E.l in Relations
      static function get_l(e : E) : [E2]{
        return if(e == null) null else [e2 | e2 : E2 in e.l];
      }[19]

      static function get_l(entities : [E]) : [E2]{
        return [e2 | e : E in entities, e2 : E2 in e.l];
      }[20]
  }
```

Figure 11 Compilation scheme for Calculate-on-Read implementation strategy.

Calculate on Read Properties. The compiled Calculate-on-Read programs have the following properties: (1) derived value reads are consistent, (2) transactions might fail, and (3) cyclic derived values cause a stack overflow exception at runtime.

Derived value consistency is based on database transactions. HTTP requests see all changes to base data from previous requests, and no changes to base data from concurrent requests. They compute the derived values, so these are consistent. The database performs optimistic locking, consequently transactions with concurrent edits to the same values are rejected. A cycle in the static dependency graph, such as group 6 in Figure 9, can admit a cyclic attribute value definition (for example a submission being a child of itself, and its grade being the average of its child grades). Such a cyclic derived value cannot be computed. The generated code will keep recursing into the getters until stack space is exhausted.

Calculate on Write. Figure 12 defines the Calculate-on-Write compilation scheme. The Calculate-on-Write compilation scheme builds on the Calculate-on-Read compilation scheme, only mentioning the new or changed WebDSL code. The general idea for Calculate-on-Write is caching all derived values, and incrementally maintaining the cached values on writes (like materialized views [14]). Updating a derived value can lead to having to update other derived values. This behavior is realized by dirty flagging (and updating) all dependent attributes on updating an attribute or relation (like push-based reactive programming [27]). To avoid unnecessary recomputation, updates are scheduled using the topological sort of the data flow graph (like stratified Datalog [3, 13]). So, to translate IceDust with Calculate-on-Write to WebDSL, we need to generate caches, dirty flagging, and recalculation.

Derived value caches store the derived values [22,25]. The properties containing the cached derived values are managed by code keeping track of dirty values [29,30,36], and code for updating dirty values [27,28].

Dirty flagging of derived values happens when underlying values are updated. WebDSL provides `extend function` hooks to intercept calls to setters. When a setter is called, all dependent values are dirty flagged by traversing the data flow `path`s [31–34]. Attributes and relations with multiplicity `?` and `1` only dirty flag when the value changes [31], while relations with multiplicity `*` and `+` dirty flag on additions and removals [32,33]. As relations have two names, dirty flagging is done for both names. Moreover, updating a relation can also implicitly remove another relation. For example, moving a submission to a different assignment

```
bobsSubmissionToMath.assignment := logicAssignment;
```

will trigger:

```
bobsSubmissionToMath.set_assignment(logicAssignment);
mathAssignment.remove_from_submissions(bobsSubmissionToMath);
logicAssignment.add_to_submissions(bobsSubmissionToMath);
```

Recalculation of derived values [35] is performed after user code is run, and before the flush to database. The computation is scheduled statically by means of the topological sort of the connected components in the data flow graph. Within a connected component, a `while` continues computing derived values until none of the derived values is dirty anymore.

Calculate on Write Properties. This compilation scheme yields programs with the following properties: (1) the derived value reads are consistent, (2) transactions might fail, (3) cyclic derived values can cause non-termination, (4) scheduling is optimal for acyclic dependency graphs, and (5) scheduling is naive for connected components inside the dependency graph.

Consistency of derived values is based on consistency of derived values within a single HTTP request, and database concurrent transaction semantics. For any changed attribute

```
//All code from Calculate-on-Read, except generated fields for attributes.
for E in Entities
  entity E {
    for a : T m in E.attributes
      a : T (default=null)[21]

    for a : T m = e1 in E.attributes
      a : T (default=calculate_a())[22]

      function update_a() {  a := calculate_a();  }[23]

    for a : T m = e1 (default) in E.attributes
      a : T (default=null)[24]
      a_default : T (default=calculate_a())[25]

      function update_a() {  a_default := calculate_a();  }[26]

    for a : T m {= e1, = e1 (default)} in E.attributes
      static function a_update_all() {
        for(e in E.get_and_empty_a_dirty()){  e.update_a();  }
      }[27]

      static function get_and_empty_a_dirty() : {E} {
        var values := E_a_dirty; E_a_dirty := Set<E>(); return values;
      }[28]

      static function a_has_dirty() : Bool {  return E_a_dirty.length != 0;  }[29]

      static function a_flag_dirty(entities : {E}){ E_a_dirty.addAll(entities);}[30]

    for E.a -> path.a2 in DataFlow where a.multiplicity in {?,1} and E2=a2.entity
      extend function set_a(newV : T){ if(a != newV){ E2.a2_flag_dirty(path); }}[31]

    for E.l -> path.a2 in DataFlow where l.multiplicity in {*,+} and E2=a2.entity
      extend function add_to_l      (n:T){ if(l != n){ E2.a2_flag_dirty(path); }}[32]

      extend function remove_from_l(n:T){ if(l != n){ E2.a2_flag_dirty(path); }}[33]

    for E.a -> path.a2 in DataFlow where a2 : T m = e1 (default) and E2=a2.entity
      extend function set_a_default(newValue : T) {
        if(a == null && a_default != newValue){  E2.a2_flag_dirty(path);  }
      }[34]
  }

// update_derivations gets called before flush to database
function update_derivations(){
  var not_empty : Bool;
  for ConnectedComponent cc in DataFlowGraph topologically sorted
    not_empty := true;
    while(not_empty){
      for a : T m {= e1, = e1 (default)} in cc where E = a.entity
        E.a_update_all();
      not_empty := false;
      for a : T m {= e1, = e1 (default)} in cc where E = a.entity
        not_empty := not_empty || E.a_has_dirty();
    }
}[35]

for a : T m {= e1, = e1 (default)} in Attributes and E = a.entity
  request var E_a_dirty : {E} := Set<E>()[36]
```

Figure 12 Compilation scheme for Calculate-on-Write implementation strategy.

or relation, all the values that depend on it are dirty flagged and recomputed. By induction, all values that depend transitively on a changed value get dirty flagged and recomputed. Computation only stops if all dirty flags are processed. As such, for a specific HTTP request, all derived values in memory are up to date when computation terminates. Flushing to the database only succeeds if previously read data remains unchanged, guaranteeing consistency. Failing transactions occur more often in Calculate-on-Write than in Calculate-on-Read, as both the updates to base values, and the updates to derived value caches can cause conflicts.

Cyclic derived values, such as the average submission grade depending on itself, can cause non-termination. If an updated value dirty flags itself (transitively), and its new value is different, the computation loops. A diverging value causes non-termination, while a converging value is a fix point calculation. Incremental Datalog implementations guarantee termination by disallowing recursive aggregation and negation: stratified negation [3] and stratified aggregation [26]. We allow recursive aggregation, but do not guarantee termination.

Derived values should only be recomputed after all values they depend upon are already updated. With acyclic data flow graphs, topological scheduling completely removes unnecessary recomputation. With cyclic data flow graphs, topological scheduling only partially removes unnecessary recomputation: the connected components are statically scheduled, but the derived values inside a connected component are updated without scheduling.

Calculate Eventually. Figure 13 defines the Calculate-Eventually compilation scheme. The Calculate-Eventually compilation scheme builds on the Calculate-on-Write compilation scheme, only stating additions and changes. The idea is to take the dirty flags from Calculate-on-Write, but pass these on to a separate, dedicated thread, allowing the HTTP request handlers to finish early. The writes to base values will still be synchronous, but the updates to derived values will be asynchronous. So, to translate to WebDSL we need to generate code that (1) dirty flags cross-thread, and (2) updates derived values in a separate thread.

Cross-thread dirty flagging communicates dirty flags from request handlers to the updater thread. WebDSL abstracts over concurrent handling of requests by running request handlers completely separated from each other. Communication between the threads handling HTTP requests, normally, is through the database. However, the database cannot notify the updater thread, so in memory communication is required. To communicate in memory between threads in WebDSL we need `native` Java code. For each derived value attribute we generate a `ConcurrentLinkedQueue` [45], and make this queue available in WebDSL by means of a static function in a `native class` [44]. As an HTTP request is handled, derived values get dirty flagged locally (as in Calculate-on-Write). After the changes are flushed to database, the local dirty flags are communicated cross-thread. Because an entity can be mapped from the relational database to an object in memory multiple times (once per request handler), the cross-thread dirty flagging needs to communicate an entity's unique identifier (UUID) [39].

The *derived value recalculation thread* is started with WebDSL's recurring tasks mechanism [42]. Every millisecond the thread is started, if not still running. The thread performs the same calculations as Calculate-on-Write, but uses the cross-thread dirty flags [43]. It loads entities with dirty flagged derived values into memory [37], then updates derived values, and finally propagates its own local dirty flags to the cross-thread dirty flags [39].

Calculate Eventually Properties. The Calculate-Eventually programs have the following properties: (1) derived values will eventually be up to date, (2) derived value reads are not glitch-free, (3) derived value calculation can starve under load, (4) after load subsides only relevant updates are calculated, and (5) cyclic values can cause non-termination.

Eventual calculation is guaranteed by the invariant that outdated derived values are

```
//All code for Calculate-on-Write, except for update_derivations.
for E in Entities
  entity E {
    for a : T m {= e1, = e1 (default)} in E.attributes
      static function get_and_empty_a_dirty_async() : {E} {
        var queue := DirtyQueues.get_E_a_queue();  var values : {E};
        while(!queue.isEmpty()){
          values.add(loadEntity(E, UUIDFromString(queue.poll() as String)) as E);
        }
        return values;
      }[37]

      static function a_has_dirty_async() : Bool {
        return !DirtyCollections.get_E_a_queue().isEmpty();
      }[38]

      static function a_flag_dirty_async() {
        var dirty := E.get_and_empty_a_dirty();
        DirtyCollections.get_E_a_queue().addAll([v.id.toString()|v : E in dirty]);
      }[39]

      static function a_update_all_async() {
        for(e in E.get_and_empty_a_dirty_async()){  e.update_a();   }
      }[40]
  }

//flag_dirty_async is called on every request after write to database
function flag_dirty_async() {
  for a : T m {= e1, = e1 (default)} in Attributes and E = a.entity
    E.a_flag_dirty_async();
}[41]

invoke update_derivations() every 1 milliseconds[42]
function update_derivations(){
  var not_empty : Bool;
  for ConnectedComponent cc in DataFlowGraph topologically sorted
    not_empty := true;
    while(not_empty){
      for a : T m {= e1, = e1 (default)} in cc where E = a.entity
        E.a_update_all();
      flagDirtyAsync();
      not_empty := false;
      for a : T m {= e1, = e1 (default)} in cc where E = a.entity
        not_empty := not_empty || E.a_has_dirty();
    }
}[43]

native class derivations.DirtyQueues as DirtyQueues {
  for a : T m {= e1, = e1 (default)} in Attributes and E = a.entity
    static get_E_a_queue() : Queue[44]
}
```

```
public class DirtyQueues {
  for a : T m {= e1, = e1 (default)} in Attributes and E = a.entity
    private static Queue<String> E_a_queue =new ConcurrentLinkedQueue<String>();[45]
    public static Queue<String> get_E_a_queue(){  return E_a_queue;  }[46]
}
```

Figure 13 Compilation scheme for Calculate-Eventually implementation strategy.

always accompanied by a dirty flag. Dirty flags are only sent by request handling threads after their changes are flushed to the database, ensuring the updater thread never processes dirty flags without seeing the changes. During updates (the same) derived values might be dirty flagged again. To ensure new dirty flags are processed, the updater thread copies and empties the dirty flag queues before processing flags. New flags will be processed subsequently.

Glitch-freedom is not provided inside connected components, as there is no topological ordering on the instance level. Also, derived value calculation starvation happens when server load is high. However, successive changes to the same base values will not create extra dirty flags. So when the system has spare resources, it will just compute the derived values based on the latest base values, and ignore all the intermediate base values. And finally, like Calculate-on-Write, cyclic derived values can cause non-termination.

5 Evaluation

The declarative specification of derived values in IceDust allows switching implementation strategies to realize different non-functional requirements without invasive code changes. In this section we benchmark different generated implementations, to evaluate whether they are indeed able to satisfy different non-functional requirements. The benchmarks differ in (1) the read/write ratio, (2) the number of base values derived values depend upon, and (3) the number of fully unrelated derived values. The measured non-functional properties are (1) throughput of derived value reads and base value writes per second, (2) the number of failing writes per second, and (3) the response time for reading derived values and writing base values. In this section we will discuss the benchmark setup and results.

Benchmark Setup. In the case study (Section 6) we encounter derived values that depend on up to 60000 values transitively. The essence of the calculation is a tree-like structure with aggregations on every level. For the benchmark evaluation of the different implementation strategies we use a simplified model, a simple tree with an average on each level:

```
entity Node {  avgValue : Float? = avg(children.avgValue) (default)  }
relation Node.parent ? <-> * Node.children
```

The tree branches out with a factor of 10, up to 6 deep (size 1, 11, . . . , 111111).

The benchmarks consist of read and write requests. Read requests retrieve the average at the top node of the tree. Write requests update a value at a random leaf node of the tree. Benchmarks are warmed up for 10 seconds, and then measured for 15 minutes. The Siege[1] tool is used to execute the benchmark requests. It is configured to use 10 concurrent threads, which initial benchmarks indicated to be a reasonable concurrency level. If the concurrency level is too low, the computer is not using maximum resources; if it is too high, too many requests will queue up increasing response times but not improving throughput. The benchmarks were performed on an early 2013 Macbook Pro laptop with Intel Core i7 2,7Ghz, 4 cores (8 threads), and 16 GB memory. The Java servlet web application generated by WebDSL was deployed on OS X 10.11, Java 1.8.0_60, MySQL 5.6.27, and Tomcat 7.0.40.

Benchmark Results. The first two benchmarks determine the behavior in extreme workloads with only read or only write requests. Figure 14 shows that the performance for a workload of 100% decreases as the tree gets deeper. Beyond depth 4 the response takes longer than

[1] https://www.joedog.org/siege-home/

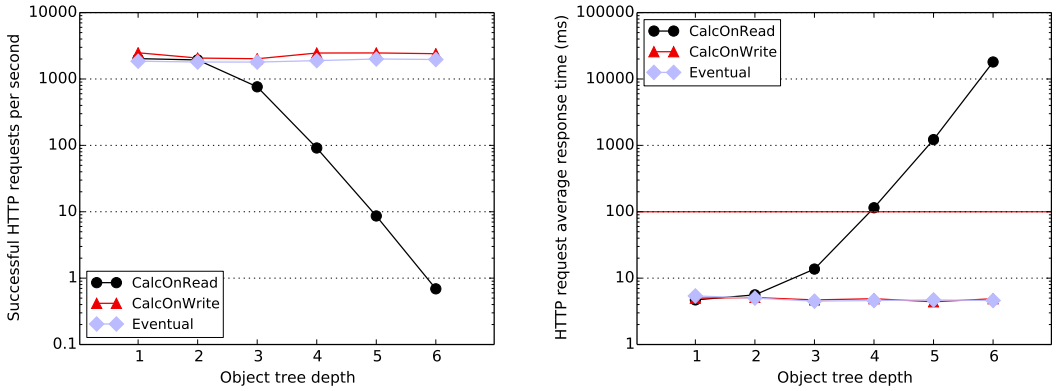


Figure 14 Read-only workload benchmark throughput (left) and latency (right).

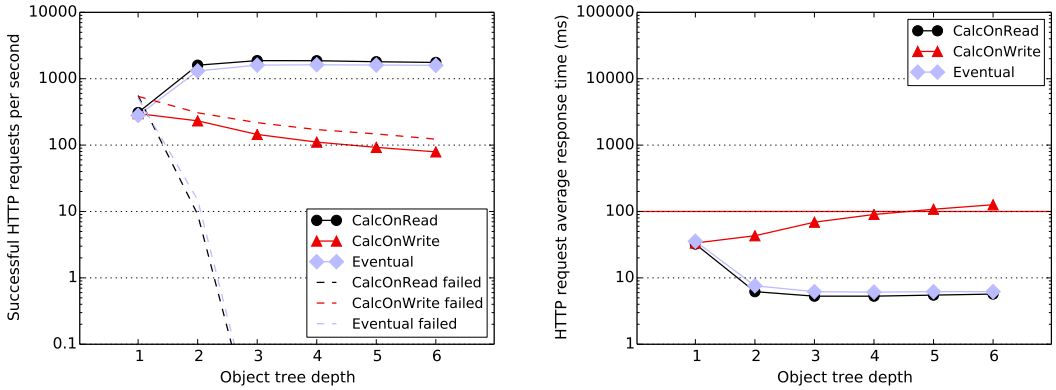


Figure 15 Write-only workload benchmark throughput (left) and latency (right).

0.1 second, and is noticeable for users. Calculate-on-Read response times increase linearly with the number of read objects, indicating that this is the limiting factor. The other implementation strategies stay at a steady high throughput with low latency, because they only retrieve a single node with a cached value on each request. The maximum throughput is around 1900 transactions per second, indicating the general overhead of the system.

The benchmark in Figure 15 shows the performance for write-only workloads. In addition to the throughput of successful requests, the failed requests are indicated by the dashed lines. When a database transaction fails due to conflicting writes, WebDSL retries it up to 3 times before failing the entire request. This improves usability for typical scenarios where a single transaction conflict may occur occasionally. Multiple subsequent failed transactions only occur in extreme situations where many concurrent requests conflict. For example, in this benchmark at tree depths 1 (1 object) and 2 (11 objects), all implementation strategies have repeated transaction failures resulting in failed requests. In general, the maximum throughput the system supports for concurrent edits on a single object is close to 300 edits per second (tree depth 1, all implementation strategies). Calculate-on-Write has many request failures (around 60%) at all tree depths, which makes the implementation unusable in practice for this use case. Calculate-on-Read and Calculate-Eventually have overall high throughput and low latency, except for the low tree depths where transaction failures occur.

Figure 16 shows the trade-off between Calculate-on-Read and Calculate-on-Write in mixed read/write ratio workloads. Calculate-on-Write suffers from many transaction failures, except for 100% reads. So, it is unusable if all base values aggregate into a single value, even

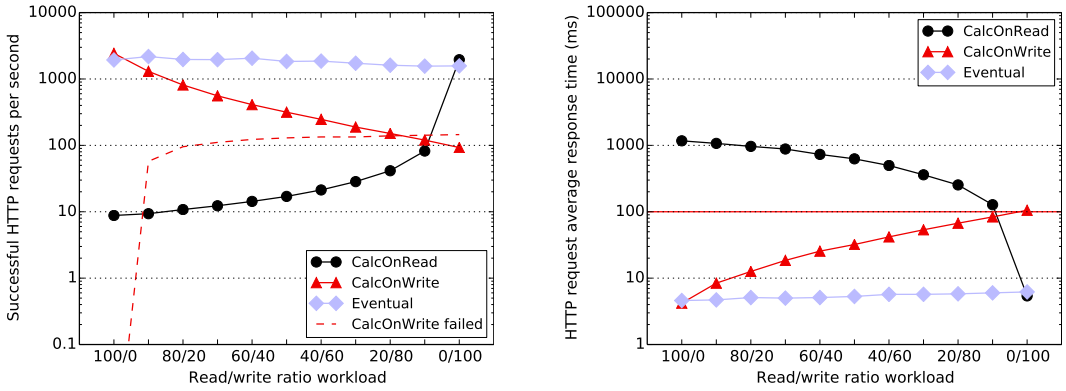


Figure 16 Varying workload benchmark throughput (left) and latency (right) with tree depth 5.

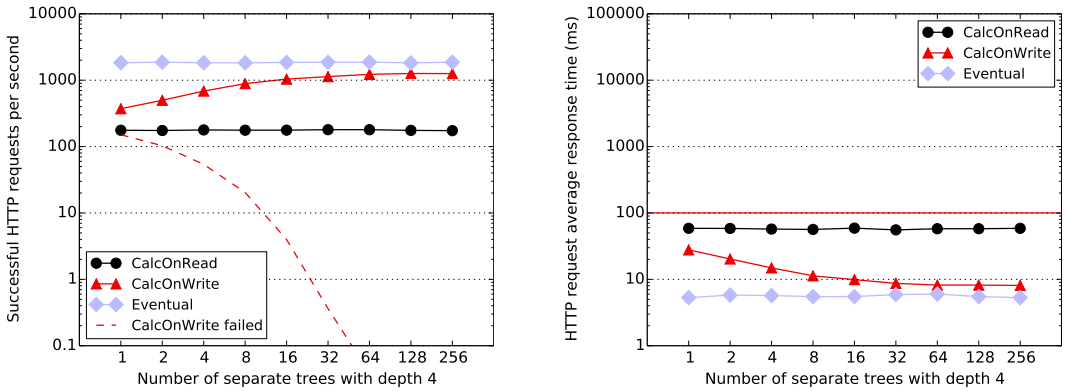


Figure 17 Separate trees benchmark throughput (left) and latency (right) with 50-50 workload.

with a small number of concurrent writes. Calculate-on-Read improves when the workload shifts from reads to writes. However, the average response time for anything except 100% writes is unacceptebly high. Calculate-Eventually has stable high throughput and low latency for all the different workload scenarios. If all base values in a system aggregate transitively into a single derived value, the only viable strategy is Calculate-Eventually.

In the final benchmark, shown in Figure 17, we investigate whether not aggregating all base values into the same derived value makes the Calculate-on-Write strategy viable. We compare the implementation strategies when there are up to 256 separate trees of depth 4. Calculate-on-Write performs better indeed in scenarios with more disconnected derived values. With 16 trees, the number of failed transactions drops below 0.5%. If consistency is desired, and derived values depend on roughly 1000 base values, the trade-off throughput-wise between Calculate-on-Read and Calculate-on-Write is at 2 separate trees. However, Calculate-on-Write still has many failing requests. Only at around 16 trees the number of failing transactions falls below 0.5%, and Calculate-on-Write becomes a viable option. When eventual calculation is acceptable, it is always the most performant solution.

Discussion. We could perform many more benchmarks (for example with other data structures than trees, or with workloads with more structure than a read/write ratio). However, the presented benchmarks show that each implementation strategy is useful in specific cases.

The form of non-functional requirements determines the form of verification required [10]. The verification for quantitative requirements is measurements, and for operational

requirements is review, test or formal verification [10]. Consistency and eventual calculation are operational requirements. We tested whether our implementations satisfy consistency and eventual calculation. In future work this can be improved with formal verification.

6 Case Study

We applied IceDust to the grading policies in a learning management system in which students can submit assignments that get graded semi-automatically. The system contains complex derived value calculations, contains a lot of data (hundreds of thousands of entities, with millions of derived values), and is subject to intense workloads on a small subset of the data. The complex derived value calculations were specified in IceDust's declarative derived value attributes, and the Calculate-Eventually strategy was used to generate an implementation. In this section we (1) reflect on the expressiveness of IceDust, based on the experiences from the case study, and (2) highlight parts of the resulting declarative specification that are quite different from the original imperative implementation.

First, let us introduce the learning management system in more detail. The system is a much more complicated version of the one introduced in Section 3. It features semi-automatic grading, programming assignments with test cases, and automatically graded multiple choice questions. Assignments are structured in a tree, and students get a weighted average for each node in the tree up to the top, which is their final grade for the course. The grading logic also includes deadlines, deadline extensions, late penalties, minimum grades, and alternative assignments. The courses, and their assignments, have statistics such as the percentage of students with a passing grade. For the largest course in the system, the statistics depend transitively on Â± 60000 individual submissions (Â± 250 students, with Â± 15 assignments per week, running for 12 weeks, and exams with multiple questions in the end).

Explicit Not Yet Calculated Values. Expressing the grading logic in IceDust forced us to look at previously implicit things. Students which did not attempt an assignment get a 0.0 on a scale of 1.0 to 10.0. The other students would get a 1.0, as grading would be triggered:

```
// only call calculateGrade if submission is attempted
function calculateGrade() {  grade := max(1.0, calculatedGrade);  }
```

The grading logic states that grades cannot be lower than 1.0, but if grading is not triggered the float default value is used. As the compiled code from IceDust detects everything that should be calculated, these grades would be changed from 0.0 to 1.0. To model assignments that are not attempted by students, `attempted` should be explicitly mentioned:

```
grade : Float = if(not attempted) 0.0 else max(1.0, calculatedGrade)
```

Explicit Stateful Calculations. The imperative code, also implicitly, kept old grades when grades where published and a newly calculated grade was lower:

```
if(assignment.statsPublic()) {  newgrade := max(oldgrade, newgrade);  }
```

In the new specification this requires an explicit self-reference:

```
grade : Float = if(public) max(grade, calculatedGrade) else calculatedGrade
```

Note that the IceDust specification only works for push-based implementations: Calculate-on-Write or Calculate-Eventually. Calculate-on-Read would throw a stack overflow exception. The sentence *'only update when grade is higher'* implies push-based calculation: the previous calculated grade needs to be cached for when a new grade becomes available. It is arguable

whether someone should express logic like this, as once grades are visible, it is not traceable anymore how a grade was calculated. This is on the border of what is expressible in IceDust.

Code Factorization Differences. In IceDust the value of an attribute is defined in a single place: the derived value expression. In imperative code, assignments to attributes can happen in multiple places, which means assignments can be distributed over `if`s:

```
if(assignment.passOne){
  passSub := disj([s.pass() | s:Submission in submissions]);
  grade   := max([s.grade() | s:Submission in submissions]);
} else{
  passSub := conj([s.pass() | s:Submission in submissions]);
  grade   := avg([s.grade() | s:Submission in submissions]);
}
```

In IceDust `if`s need to be distributed over derived value attributes:

```
grade   : Float?  = if(assignment.passOne) max(children.grade)
                    else                   avg(children.grade)

passSub : Boolean = if(assignment.passOne) disj(children.pass) // pass one
                    else                   conj(children.pass) // pass all
```

Whether the old or new specification is preferable is arguable. If the cases would be more complex than a single if it would lead to repeated code in IceDust.

Application Analysis. Finally, we made a more analytic observation: even though the data-flow graph of the specification in IceDust contains more than 100 nodes, it contains just a single connected component. Grades, weightedGrades (weighted averaging is used), pass, and child-pass are mutually recursive (like Figure 9). All other dependencies are acyclic. This system has derived value-wise just a single complex part: the grade calculation. Intuitively we already knew this, but now we can quantify this with properties of the data-flow graph.

7 Related Work

The related work is organized along language design and the three implementation techniques (Calculate-on-Read, Calculate-on-Write, and Calculate-Eventually).

Languages with Relations. There are multiple languages that feature relations as a language construct. We will cover closely related languages and highlight the differences.

Rumer [4] features first-class citizen relations with queries for navigation. IceDust relations are not first-class citizen, and navigation is through member access instead of queries. In Rumer multiplicities can be specified in constraints which are enforced at runtime, while in IceDust these are part of the type system. Rumer does not support derived value attributes, but queries can be used to specify Calculate-on-Read derived values. Finally, Rumer is an imperative in-memory language, while IceDust is declarative and persists its data.

RelJ [5] also features first-class citizen relations. In RelJ participants of relations do not have names, so navigation is positional (using `from` and `to`). RelJ features only multiplicity upper bounds, no lower bounds. These multiplicities are enforced at runtime: either by throwing exceptions, or by implicitly removing previous relations. RelJ does not feature derived value attributes, and RelJ is an imperative in-memory language like Rumer.

Relations [18] features multiplicities as part of the type system and derived value attributes like IceDust. Its derived values are, however, only Calculate-on-Read. Relations is declarative,

like IceDust, but its data is only in memory, not persistent. Relations in this language are first-class citizen like Rumer and RelJ, but feature navigation through member access. IceDust relations are not first-class citizen, but feature the same member access navigation.

Alloy [21] is a language for bounded model checking which features language constructs similar to IceDust: bidirectional relations, multiplicities, and derived values. Alloy is more expressive than IceDust: it features n-ary relations, and its derived values specify derived relations (as opposed to derived attribute values). However, Alloy's bounded model checker only works on small data sets, and primitive values (only integers in Alloy) should be avoided as they blow up the state space. IceDust, on the other hand, supports derived values over arbitrary primitive values (int, string, float, datetime, and boolean), and admits efficient implementation strategies applied to large data sets. To compute derived values in large data sets from an Alloy specification, Alloy would need an operational semantics not based on bounded model checking or SAT solving. An approach for an operational semantics for Alloy was proposed in [9], but this approach is not complete. As Alloy has much greater expressive power (first-order logic), we also expect such an Alloy operational semantics to not be efficient. Finally, another difference is that in IceDust the multiplicities are checked in the type system, while in Alloy these are only checked during bounded model checking.

Calculate on Read. We do not cover Calculate-on-Read extensively, as it is the default implementation for many formalisms. We cover only the object-oriented approaches.

Object-oriented languages lend themselves for various Calculate-on-Read optimization techniques. Wiedermann and Cook take imperative code with for loops and if statements and convert those to SQL queries [37]. Their approach is similar to our work in that it operates on persistent objects by means of an object-relational mapper. Also their approach for analyzing dependencies is similar: path-based abstract interpretation. They optimize imperative code that can be expressed as queries. Our approach, on the other hand, treats code that cannot be expressed as queries, recursive aggregation. The Java Query Language (JQL) adds queries to Java [38]. The rationale for queries is that these are more succinct to write, and more efficient than nested for loops. JQL has been incrementalized, we will cover this in the next subsection. This paper adds over these approaches the possibility to easily switch to an incremental or eventual calculation implementation strategy.

Incremental Computation (Calculate on Write). Incremental computation is present in many fields in computer science. We relate our Calculate-on-Write implementation scheme to existing incremental approaches.

Materialized views in relational databases can be incrementally maintained [14]. Recursion and stratified aggregation can be supported [15]. Stratified aggregation does not admit recursive aggregation. (See next paragraph for relaxations of stratified aggregation in logic databases.) Switching between implementation strategies in relational database also do not require invasive code changes: the definitions for materialized and non-materialized views are identical. Relational databases do, however, not support eventually-calculated views.

Logic Databases or Deductive Databases are a more expressive than relational databases. Logic Databases support stratified aggregation like relational databases [26]. Since stratified aggregation does not support recursive aggregation, more relaxed notions of aggregation have been introduced, such as Monotone Aggregation [29]. Monotone Aggregation has also been incrementalized [28]. A recent survey [13] states that at present, the Datalog community seems not to have converged on any of the proposed semantics for aggregation through recursion. This means that in practice recursive aggregation is often not supported. For example LogiQL [12], the language of LogicBlox, does not support recursive aggregation.

Functional reactive programming (FRP) [8], with for example REScala [30], Scala.React [23], or i3QL [25], provides incremental computation. Calculate-on-Read style code wrapped with FRP libraries behaves as Calculate-on-Write. FRP abstractions provide single-threaded, in-memory derived values. In contrast, IceDust provides concurrent, persistent derived values.

Spreadsheets provide incremental computation. The data structure in a spreadsheet is a 2d grid. IceDust's data structure is an object graph. Moreover our object graph is typed, while spreadsheets are free form. Spreadsheets do mostly have an implicit structure [19]. IceDust with Calculate-on-Write can be seen as a structured spreadsheet without a 2d grid.

Object-oriented programs can also be incrementalized. Incremental Updates for Materialized OQL views [11] proposes to generalize incremental view maintenance from relational databases to support the Object Query Language (OQL) as view definition language. MOVIE [2] develops this work further. They also provide an overview of relational incremental view maintenance implementations, with either a relational or an object-oriented surface syntax. These approaches, even though some have an object-oriented surface syntax, are part of the relational paradigm (with the limitations previously mentioned for materialized views).

The Java Query Language is incrementalized [39]. Their benchmarks show, like ours, that for different read-write ratio workloads the incremental or calculate-on-read solution offers better performance. Demand-Driven Incremental Object Queries [22] improves over JQL by using auxiliary indices for incrementality. Similar to [37] they transform imperative code to a relational calculus. But instead of performing relational queries like [37] they use the relational model to generate code that incrementally maintains the caches. Our approach uses path-based abstract path interpretation instead of a relational calculus to generate maintenance code. Both of the above approaches slightly differ in use cases from our approach: they target set membership of objects e.g. whether an object belongs to a set specified by a query, while our approach targets derived value attributes.

Graph queries can be incrementally evaluated in IncQuery [34]. IncQuery's data structure is a graph, like ours, but its goal is to pattern match. Our approach does not support pattern matching on graph structures, rather it computes derived attribute values.

Attribute grammars feature a declarative style of specifying derived values. Attribute grammars can also be incrementally computed [7]. As attribute grammars only support trees, one could look at reference attribute grammars to support full blown graphs [31]. Reference attribute grammars do support graph structures, but there is a clear distinction between the tree, and the derived graph edges. In our approach the graph is the basis. Fitting our data models onto attribute grammars would require extracting a spanning tree, and deriving the other edges. In this process we would lose the correspondence to the data flow graph, and derived edges would become dynamic dependencies, which would complicate scheduling.

Self-adjusting computation [1] does not cover a single programming paradigm as it features multiple languages (including SLf, for functional programming, and SLi, for imperative programming). Self-adjusting computation automatically transforms a Calculate-on-Read style program to a Calculate-on-Write style program. Our approach does not take Calculate-on-Read as basis, but instead provides a declarative language to express derived values.

Eventual Calculation. The code generated by the Calculate-Eventually implementation scheme makes derived values of attributes eventually consistent with base values. We cover existing work on eventual (or asynchronous) computation of derived values in this subsection.

Event and Actor programming, with for example Akka [16] or RX [24], provide an asynchronous update mechanism for calculating derived values. Updates to derived values are asynchronous, meaning that there is no consistent view of base values and derived values

at the same time. As such, these do not provide consistency, like the code produced by our Calculate-Eventually implementation strategy.

Eventual consistency for distributed data also features eventual calculation, but is unrelated. As a recent survey on Eventual Consistency states: "shared data is updated at different replicas, updates are transmitted asynchronously, and conflicts are resolved consistently" [6]. Our approach does not have different replicas of data, there is a single database. Our approach does not have asynchronous updates, the update is synchronous as a HTTP response is only sent after the transaction in the database is completed. And finally, our approach does not have conflicts during the calculation of derived values, as the base values define unambiguously what the derived values of attributes should be.

8 Conclusion

Data modeling with declarative derived value attributes in IceDust allows deferring the decision about implementation strategy from implementation to compilation time, and allows switching strategies without invasive code changes. We have demonstrated that these different strategies provide different non-functional properties, so that a specific strategy can be chosen to realize certain non-functional requirements. Finally, a case study indicated our approach is useful for expressing derived values of systems used in practice.

In future work, we would like to explore more implementation strategies, such as transitive dirty flagging on writes with recalculation on reads, or eventually calculated with flags indicating whether the values are up to date or not. We also would like to explore more flexibility in implementation strategies by allowing composition of different strategies, and live switching between strategies. A type system should restrict compositions to only sound ones: consistent values cannot depend on eventually calculated values, and calculate on write values cannot depend on calculate on read values. Finally, we would like to guarantee termination by specifying non-circular relations and runtime non-circularity checking.

References

1 Umut A. Acar. Self-adjusting computation: (an overview). In *PEPM*, pages 1–6, 2009. doi:10.1145/1480945.1480946.

2 M. Akhtar Ali, Alvaro A. A. Fernandes, and Norman W. Paton. Movie: An incremental maintenance system for materialized object views. *DKE*, 47(2):131–166, 2003. doi:10.1016/S0169-023X(03)00048-X.

3 Krzysztof R Apt, Howard A Blair, and Adrian Walker. *Towards a theory of declarative knowledge*. IBM Thomas J. Watson Research Division, 1986.

4 Stephanie Balzer. *Rumer: a Programming Language and Modular Verification Technique Based on Relationships*. PhD thesis, ETH, Zürich, 2011.

5 Gavin M. Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *ECOOP*, pages 262–286, 2005. doi:10.1007/11531142_12.

6 Sebastian Burckhardt. Principles of eventual consistency. *FTPL*, 1(1-2):1–150, 2014. doi:10.1561/2500000011.

7 Alan J. Demers, Thomas W. Reps, and Tim Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *POPL*, pages 105–116, 1981. doi:10.1145/567532.567544.

8 Conal M. Elliott. Push-pull functional reactive programming. In *haskell*, pages 25–36, 2009. doi:10.1145/1596638.1596643.

9 Theophilos Giannakopoulos, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Towards an operational semantics for alloy. In *FM*, pages 483–498, 2009. doi:10.1007/978-3-642-05089-3_31.

10 Martin Glinz. Rethinking the notion of non-functional requirements. In *WCSQ*, pages 55–64, 2005.

11 Dieter Gluche, Torsten Grust, Christof Mainberger, and Marc H. Scholl. Incremental updates for materialized oql views. In *DOOD*, pages 52–66, 1997. doi:10.1007/3-540-63792-3_8.

12 Todd J. Green. Logiql: A declarative language for enterprise applications. In *PODS*, pages 59–64, 2015. doi:10.1145/2745754.2745780.

13 Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *FTDB*, 5(2):105–195, 2013. doi:10.1561/1900000017.

14 Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *DEBU*, 18(2):3–18, 1995.

15 Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, 1993. doi:10.1145/170035.170066.

16 Munish Gupta. *Akka essentials*. Packt Publishing Ltd, 2012.

17 Terry Halpin. Object-role modeling (orm/niam). In *Handbook on architectures of information systems*, pages 81–103. Springer, 2006. doi:10.1007/978-3-662-03526-9_4.

18 Daco Harkes and Eelco Visser. Unifying and generalizing relations in role-based data modeling and navigation. In *SLE*, pages 241–260, 2014. doi:10.1007/978-3-319-11245-9_14.

19 Felienne Hermans, Martin Pinzger, and Arie van Deursen. Automatically extracting class diagrams from spreadsheets. In *ECOOP*, pages 52–75, 2010. doi:10.1007/978-3-642-14107-2_4.

20 Shan Shan Huang and Yannis Smaragdakis. Expressive and safe static reflection with morphj. In *PLDI*, pages 79–89, 2008. doi:10.1145/1375581.1375592.

21 Daniel Jackson. Alloy: a lightweight object modelling notation. *TOSEM*, 11(2):256–290, 2002. doi:10.1145/505145.505149.

22 Yanhong A Liu, Jon Brandvein, Scott D Stoller, and Bo Lin. Demand-driven incremental object queries. *arXiv preprint arXiv:1511.04583*, 2015.

23 Ingo Maier and Martin Odersky. Higher-order reactive programming with incremental lists. In *ECOOP*, pages 707–731, 2013. doi:10.1007/978-3-642-39038-8_29.

24 Erik Meijer. Reactive extensions (rx): curing your asynchronous programming blues. In *CUFP*, page 11, 2010. doi:10.1145/1900160.1900173.

25 Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. i3ql: language-integrated live data views. In *OOPSLA*, pages 417–432, 2014. doi:10.1145/2660193.2660242.

26 Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *VLDB*, pages 264–277, 1990.

27 H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, 2002. doi:10.1145/581690.581695.

28 Raghu Ramakrishnan, Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Efficient incremental evaluation of queries with aggregation. In *SLP*, pages 204–218, 1994.

29 Kenneth A. Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In *PODS*, pages 114–126, 1992. doi:10.1145/137097.137852.

30 Guido Salvaneschi, Gerold Hintz, and Mira Mezini. Rescala: bridging between object-oriented and functional style in reactive applications. In *AOSD*, pages 25–36, 2014. doi:10.1145/2577080.2577083.

31 Emma Söderberg and Görel Hedin. Incremental evaluation of reference attribute grammars using dynamic dependency tracking. Technical Report 98, Lund University, 2012.

32 Friedrich Steimann. Content over container: object-oriented programming with multiplicities. In *OOPSLA*, pages 173–186, 2013. doi:10.1145/2509578.2509582.

33 Friedrich Steimann. None, one, many - what's the difference, anyhow? In *SNAPL*, pages 294–308, 2015. doi:10.4230/LIPIcs.SNAPL.2015.294.

34 Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. Incquery-d: A distributed incremental model query framework in the cloud. In *MoDELS*, pages 653–669, 2014. doi:10.1007/978-3-319-11653-2_40.

35 Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAMCOMP*, 1(2):146–160, 1972. doi:10.1137/0201010.

36 Eelco Visser. WebDSL: A case study in domain-specific language engineering. In *GTTSE*, pages 291–373, 2007. doi:10.1007/978-3-540-88643-3_7.

37 Ben Wiedermann and William R. Cook. Extracting queries by static analysis of transparent persistence. In *POPL*, pages 199–210, 2007. doi:10.1145/1190216.1190248.

38 Darren Willis, David J. Pearce, and James Noble. Efficient object querying for java. In *ECOOP*, pages 28–49, 2006. doi:10.1007/11785477_3.

39 Darren Willis, David J. Pearce, and James Noble. Caching and incrementalisation in the java query language. In *OOPSLA*, pages 1–18, 2008. doi:10.1145/1449764.1449766.

Magic with Dynamo – Flexible Cross-Component Linking for Java with Invokedynamic*

Kamil Jezek[1] and Jens Dietrich[2]

1 **NTIS – New Technologies for the Information Society**
Faculty of Applied Sciences, University of West Bohemia
Pilsen, Czech Republic
kjezek@kiv.zcu.cz

2 **School of Engineering and Advanced Technology, Massey University**
Palmerston North, New Zealand
j.b.dietrich@massey.ac.nz

Abstract

Modern software systems are not built from scratch. They use functionality provided by libraries. These libraries evolve and often upgrades are deployed without the systems being recompiled. In Java, this process is particularly error-prone due to the mismatch between source and binary compatibility, and the lack of API stability in many popular libraries. We propose a novel approach to mitigate this problem based on the use of `invokedynamic` instructions for cross-component method invocations. The dispatch mechanism of `invokedynamic` is used to provide on-the-fly signature adaptation. We show how this idea can be used to construct a Java compiler that produces more resilient bytecode. We present the dynamo compiler, a proof-of-concept implemented as a `javac` post compiler. We evaluate our approach using several benchmark examples and two case studies showing how the dynamo compiler can prevent certain types of linkage and stack overflow errors that have been observed in real-world systems.

1998 ACM Subject Classification D.1.5 Object-oriented Programming D.2.13 Reusable Software D.3.4 Processors

Keywords and phrases Java, compilation, linking, binary compatibility, invokedynamic

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.12

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.2.1.5`

1 Introduction

Java and similar languages support dynamic linking to enable programs to use libraries that have been compiled separately. This makes it possible to decouple the lifecycles of a system and the libraries it uses, and to deploy new versions of libraries without reinstalling the entire system. This is particularly important for server-based systems with high availability requirements, such as services running in "24/7" mode and systems with service level agreements (SLAs). For this to work, library evolution must follow certain compatibility rules. In particular, the APIs used by the system or other libraries have to remain stable. Unfortunately, APIs do change when libraries evolve [13, 6, 11]. In Java programs, this can result in linkage errors indicating that references to library code cannot be resolved.

* This work was partially supported by Oracle Labs, Australia.

30th European Conference on Object-Oriented Programming (ECOOP 2016).
Editors: Shriram Krishnamurthi and Benjamin S. Lerner; Article No. 12; pp. 12:1–12:25

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

For instance, if the signature of a library method changes, attempts to link and run a client program using this method generate a `NoSuchMethodError`. Some other language and execution platform combinations such as C# / CLR exhibit similar behaviour.

The relevant part of the signature used to resolve method references is called the descriptor [26, sect 4.3.3]. The descriptor consists of the parameter and return types without generic type parameters. Changes to descriptors cause linkage errors. This leads to a behaviour that is very different from the compiler that reasons about the type hierarchy, associations between primitive and wrapper types, and narrowing and widening rules. In other terms, there are different types of compatibility [8]: *source compatibility* is used by the compiler when a system is compiled against a set of libraries, while *binary compatibility* is used when a system is linked against libraries that may have been compiled separately. The notion of binary compatibility is explicitly defined in the Java Language Specification with respect to linking [22, sect 13].

There are a number of empirical studies all indicating that this leads to problems [13, 6, 11, 32, 33]. It turns out that binary compatibility issues are surprisingly common when libraries evolve, and the majority of developers lack understanding of these issues [12].

There are several possible approaches to tackle this problem. First, meta-object protocols and patterns could be used. For instance, in the presence of protocols like Smalltalks `doesNotUnderstand` [21] or Ruby's `method_missing` [18], adapters can be easily integrated into library code. But this requires that the respective classes override `doesNotUnderstand`. And anyway, such a protocol is not available in Java. The second option is to change the runtime (i.e. the VM): either by instrumenting code that is being loaded or by changing the linking process in the virtual machine itself. This is possible, but expensive and invasive - the runtimes must be configured accordingly, and the instrumentation imposes a performance penalty. Moreover, the same code is sometimes accessed from different contexts (other libraries or applications), with different lifecycle dependencies on the library containing the code that is being adapted. Instrumenting library code does not support such scenarios. The third option is to access objects instantiating library classes via proxies and to use reflection within these proxy classes to resolve methods. This has two disadvantages: the proxies must either be generated or a dynamic proxy pattern must be used which requires some type abstraction. Furthermore, the use of reflection is slow.

In this paper, we suggest a simple yet elegant solution to this problem that tries to combine the advantages of the other methods discussed. Our approach is based on the idea of replacing existing invoke instructions of library methods ("cross-component invocations") by `invokedynamic` instructions. The runtime bootstrap mechanism we propose mimics the behaviour of a solution based on a `doesNotUnderstand`-like protocol. While we use a reflection protocol to locate target methods, the use of the protocol based on `invokedynamic` allows us to avoid much of the runtime overhead of traditional reflection. Bytecode is manipulated at compiletime and not when classes are loaded, thus avoiding runtime performance and configuration overhead. Also, the generated bytecode can be further optimised and analysed by the standard Java runtime (JIT, HotSpot). Due to the use of `invokedynamic`, the bytecode produced by our method is less prone to linkage errors than the bytecode produced by the standard Java compiler.

Our contributions are as follows:

- We present the *dynamo enhancer*, a bytecode manipulation framework that can be used to replace invocation instructions.
- We present the *dynamo compiler*, a Java compiler based on the dynamo enhancer.
- We present the *DynamoDSL*, a lightweight declarative language that can be used to specify when to replace invocations.

- We present a set of benchmarks used to measure the compile- and runtime performance overhead of dynamo.
- We present a case study that shows how the use of the dynamo compiler can prevent linkage errors that occur in the evolution of the *jasperreports* and *jfreechart* open source libraries.
- We present a second case study that shows how the use of the dynamo compiler can prevent stack overflow errors that can occur when methods are overridden with covariant return types.

The rest of this paper is structured as follows: first we discuss related work in Section 2 followed by a background in Section 3 where we discuss some key concepts used later. In Section 4, we present the actual compiler and its components, followed by a discussion of the runtime behaviour in Section 5. This is followed by Section 6 discussing benchmarks used for quality assurance and performance assessment, and two case studies in Section 7 extracted from problems encountered with real world systems. We finish our contribution with a brief conclusion.

2 Related Work

Our work addresses issues related to binary compatibility. The study of binary compatibility goes back to the work by Forman et al. [19], in the context of IBM's SOM object model. For Java, binary compatibility is formally defined in the language specification [22, sect 13]. Drossopoulou et al. have proposed a formal model of binary compatibility in [17].

Our work is motivated by issues that result from inconsistencies between source, binary and to some extent behavioural compatibility. These issues have been catalogued and studied by several authors, including des Rivières [10], Dietrich et al. [11, 12] and Raemakers et al. [32]. In particular, they include empirical studies [11, 32] showing that binary compatibility issues occur in practice when programs and libraries used by these programs evolve independently. Dig and Johnson [13] have conducted an API evolution case study on five real world systems (*struts*, *eclipse*, *jhotdraw*, *log4j* and a commercial application). They found that the majority of API-breaking changes were caused by refactoring, but did not distinguish between different types of compatibility. Mens et al. [27] have studied the evolution of Eclipse (from version 1.0 to version 3.3). The focus of this study was to investigate the applicability of Lehmann's laws of software evolution [25]. They found significant changes. Cosette and Walker have studied the evolution of APIs on a set of five Java open source programs [6]. They focused on generating change recommendation techniques that could then be used to give developers advice on how to refactor client code in order to adapt to API changes.

Several authors looked into how binary compatibility problems in Java programs can be avoided. Our work is somehow similar to binary component adaptation (BCA) by Keller et al. [24] as bytecode is modified in order to overcome certain compatibility problems. However, there are important differences: (1) BCA does not support changes to method descriptors, (2) changes must be specified by the user in the form of delta files while our approach is completely automated (3) BCA modifies libraries (components) whereas we transform the program itself (4) BCA can alter types and their relationships and is therefore rather invasive as it changes the semantics of client programs using reflection (for instance, when interfaces are added to classes, and the client program uses `instanceof` guards)[1]

[1] In a trivial sense, the dynamo compiler also changes the semantics of programs as method invocations that used to result in errors succeed after compilation with dynamo. But this effects are intended and local as they only impact the objects aliased at the call site, whereas changing types will affect other, unrelated parts of the program as well.

Other related works include Dmitriev's proposal to use binary compatibility checks in order to optimise build systems [16], Barr and Eisenbach's rule-based tool to compare library versions in order to detect changes that cause binary incompatibility [2], and refactoring-based approaches by Dig et al. [14] and Savga and Rudolf [7], aiming at generating a compatibility layer that ensures binary compatibility when used libraries evolve. Corwin et al. [5] have proposed a modular framework that uses a higher level API on top of the Java classpath architecture. This is somehow similar to how the OSGi framework [40] operates.

Our work relies on the use of the `invokedynamic` instruction. One of the key features of `invokedynamic` is that it adds reflection-like features to the language without incurring the performance overhead imposed by the use of traditional reflection. The low performance overhead has been confirmed by several authors including Kaewkasi [23] and Ortin and Conde [29, 3].

Within the Java standardisation process, JEP 276: Dynamic Linking of Language-Defined Object Models [38] is related to this research. The focus of JEP 276 is on supporting the compilation of object expressions often used in applications that require templating, while we suggest different compilation for standard Java call sites. The JEP 276 proposal does explicitly state that it does "not wish to provide linking semantics for operations for any single programming language or execution environment for any such language".

There is one open source project we are aware of with a similar aim – Kohsuke Kawaguchi's Bridge Method Injector [2]. The idea behind this project is to address one particular evolution problem we also consider - source-compatible changes of the return type. This is achieved by generating additional bridge methods in a post compilation step, a technique also used by the compiler when encountering co-variant return types. This will be discussed in more detail in section 7.2. The bridge method injector relies on an annotation that has to be added by the author of the library that evolves. This requires that the author understands the effects this change has on client code, and this might not always be the case [12]. On the other hand, our approach is completely automated and covers a wider range of evolution patterns.

This work is part of a wider trend towards self-adaptive software [36, 37].

3 Background

3.1 Binary vs Source Compatibility

The compiler and the linker use sets of rules to establish whether a method invocation can be resolved. These rule sets define *source compatibility* (in the case of the compiler) and *binary compatibility* (in the case of the linker). As far as Java (version 8) is concerned, binary compatibility is generally stricter than source compatibility as the compiler can reason about subtype relationships, the associations between primitive types and their respective wrapper types via auto-boxing and unboxing conversions [22, sects 5.1.7, 5.1.8] and invocations of static methods from non-static contexts [3].

For instance, consider Listing 1[4]. This example has a client program with a class `Main` invoking `Foo.get()` defined in a library that has two versions `lib-v1.jar` and `lib-v2.jar`. `Main` can be compiled against both versions of the library. However, when `Main` is compiled against `lib-v1.jar` and then executed with `lib-v2.jar`, a linkage error

[2] `http://bridge-method-injector.infradna.com/`

[3] There are some exceptions to this rule – situations where a program is binary but not source compatible with a library. For instance, the use of erasure can cause such scenarios.

[4] Package declarations and imports are omitted.

```
// lib-v1.jar
public class Foo {
  public static java.util.Collection get() {return new java.util.ArrayList();}
}
// lib-v2.jar
public class Foo {
  public static java.util.List get() {return new java.util.ArrayList();}
}
// client program
public class Main {
  public static void main(String[] args) {
     java.util.Collection coll = Foo.get();
     System.out.println(coll);
  }
}
```

Listing 1 Specialising the return type of a method.

```
// lib-v1.jar
public class Foo {
  public static void bar(int i) {System.out.println(i);}
}
// lib-v2.jar
public class Foo {
  public static void bar(Integer i) {System.out.println(i);}
}
// client program
public class Main {
  public static void main(String[] args) {new Foo().bar(42);}
}
```

Listing 2 Boxing of a parameter type.

(`NoSuchMethodError`) is thrown as the descriptor `get()Ljava/util/Collection;` found at the call site does not match the new descriptor `get()Ljava/util/List;` and can therefore not be resolved.

Listing 2 is similar, but this time the sole parameter type of `bar()` is changed from `int` to `java.lang.Integer`. The compiler deals with this situation by applying auto-boxing. But since the descriptor changes from `bar(I)V` to `bar(Ljava/lang/Integer;)V`, this change is not binary compatible.

Finally, consider Listing 3. `Main` still compiles against `Foo` in `lib-v2.jar`, but this time the compiler has to apply two adaptation rules: auto-boxing and type generalisation. The type of the invocation must also be changed as the `static` modifier has been added to `bar`.

Evolution patterns and their impact on binary compatibility have been catalogued by ddes Rivières [10]. Some of these problems could be avoided if linking was more consistent with compilation. In particular, we are interested in the following evolution patterns that are source compatible, but not binary compatible:

1. the specialisation of a reference return type of a method
2. the narrowing of a primitive return type of a method
3. the generalisation of a reference parameter type of a method

```
// lib-v1.jar
public class Foo {
  public void bar(int i) {System.out.println(i);}
}
// lib-v2.jar
public class Foo {
  public static void bar(Object i) {System.out.println(i);}
}
// client program
public class Main {
  public static void main(String[] args) {new Foo().bar(42);}
}
```

Listing 3 Complex (but still compatible) evolution.

4. the widening of a primitive parameter type of a method
5. replacing the primitive return or parameter type of a method by the respective wrapper type
6. replacing the wrapper return or parameter type of a method by the respective primitive type
7. changing a non-static method to a static method
8. changing a class to an interface or vice-versa
9. some combinations of any number of evolution patterns from this list[5]

3.2 The invokedynamic Instruction

Prior to version 7, Java used four different bytecode instructions for method invocation. First, `invokestatic` is used to call static methods. Static methods are resolved at compile time. Next, `invokespecial` is used for special cases including constructor and private method invocations and invocations via `super`. Finally, `invokevirtual` and `invokeinterface` are used to call non-static, non-private methods. Dynamic dispatch is used here, i.e. the method which is invoked is only computed at runtime based on the actual type of the receiver.

Starting with Java 1.7, `invokedynamic` was added to the instruction set [35]. The motivation was to give programmers more control over the dispatch process, in particular to facilitate the implementation of dynamic languages like Ruby on the JVM [28]. The Java 7 JVM supports the `invokedynamic` instruction, it is not emitted by the Java 7 compiler. The Java 8 compiler uses `invokedynamic` to compile lambdas.

With `invokedynamic`, the method reference is resolved by means of a *bootstrap method*. This user-implemented method is then used to locate the actual method being invoked. This is fast as the bootstrap method is only invoked by the linker during the resolution phase [26, 5.4.3.6] and following method invocations skip the bootstrap process. At this point it is possible to implement adapters.

The bootstrap method represents the target method as an instance of a `java.lang.invoke.MethodHandle`. The method handles support transformations that can be used to achieve a linking behaviour that is similar to the behaviour of the compiler. This mapping behaviour is exposed in the API of `MethodHandle` by the `invoke` method (as opposed to the `invokeExact` method). The overall goal of this work is to use this API to perform

[5] See also section 5 for a discussion on which of those combinations are supported.

appropriate type conversions to match the call site with the method found in a library that may have evolved. The difficult part is to lookup the best method which is suitable for re-typing.

4 Compilation

The mechanism we are proposing is based on the idea of swapping invoke instructions in bytecode. Our aim is to do this at compile time, so that the bytecode produced by the compiler can be transparently used at runtime with minimal additional configuration needed. However, there is another use case: to retrofit existing code only available in compiled form. We address this use case first and introduce the *dynamo enhancer* - the tool that performs the bytecode transformations. We can then use the enhancer to design the actual *dynamo compiler*, implemented as a post compiler / wrapper around the enhancer and the standard Java compiler. This design gives the dynamo compiler almost instantly the quality attributes needed for real world application. In particular, most Java compiler optimisations are available to us. An additional benefit of this design is that the enhancer can be used as a post compiler for other compilers emitting JVM bytecode, such as `scalac`. In the final part of this section, we discuss *Dynamo DSL*, a lightweight domain-specific language that can be used to customise the dynamo-specific part of the compilation via a simple command line argument.

4.1 The Dynamo Bytecode Enhancer

The bytecode enhancer is used to transform the bytecode emitted by a standard Java compiler, and replaces selective `invokestatic`, `invokevirtual`, `invokeinterface` and `invokespecial` instructions (from hereto referred to as *classical invocations*) by `invokedynamic`. The intention of the selection is to only replace invocations where the target is located in a library that may have a separate update cycle. This is achieved through *filters*, described in more detail below.

Any classical invocation can be expressed by a tuple $t = (opcode, C, m, desc)$ where *opcode* represents one of the classical invocations, C is an owner class containing the method, m is the method's name and *desc* is the methods descriptor consisting of the formal parameter types and the return type of the method. To convert a method invocation to `invokedynamic`, the original instruction from the tuple t must be changed to fit the `invokedynamic` call site specifier [26, sect 4.7.23]. In particular, a reference to a bootstrap method must be provided that is then used at link time to locate the actual target of the invocation.

Given a tuple t containing call site information, a dynamic call site can be created as follows. An `invokedynamic` instruction is created that has an index pointing to a constant pool entry of the type `CONSTANT_InvokeDynamic`. This entry defines a bootstrap method with the following three parameters: (1) the `MethodHandles.Lookup` factory used to locate and check access to methods, (2) the method name, (3) and a `MethodType` representing the descriptor. Any number of additional user parameters may follow.

The JVM Specification provides information about descriptors used by the classical invocations and the descriptors required by method handles [26, sect 5.4.3.5]. By combining these two parts of the specification, we infer the transformation rules listed in Table 1.

As discussed earlier, C and m represent the method owner type and its name, respectively. Furthermore, $A*$ is a set of input parameter types, T represents the return type, and V represents the `void` type (used in the descriptors of constructors). For virtual and interface invocations, the owner type is prepended to the list of argument types in the target descriptor.

Table 1 Translation from method descriptors to method handles.

opcode	source C,m,desc	target desc	parameters
`invokevirtual`	`C,m,(A*)T`	`(C,A*)T`	
`invokestatic`	`C,m,(A*)T`	`(A*)T`	`C`
`invokeinterface`	`C,m,(A*)T`	`(C,A*)T`	
`invokespecial` (`new`)	`C,<init>,(A*)V`	`(A*)C`	
`invokespecial` (`super`)	`C,m,(A*)T`	`(C,A*)T`	

```
1 bipush    10
2 istore_1
3 ...    // new lib/Foo()
4 iload_1
5 invokevirtual lib/Foo.setValue:(I)V
```

Listing 4 Bytecode for `new Foo().setValue(10)` before enhancement.

```
1 bipush    10
2 istore_1
3 ...    // new lib/Foo()
4 iload_1
5 invokedynamic setValue:(Llib/Foo;I)V
```

Listing 5 Enhanced bytecode for code in listing 4.

This is necessary to check the type of the `this` reference passed as the first argument to the respective method handle. For constructors, C becomes the return type. The descriptors of method handles for static invocations do not contain owner types at all. For this reason, we pass the owner type as an additional, user defined, parameter to the bootstrap method.

The translation of the first three instructions in Table 1, `invokevirtual`, `invokestatic` and `invokeinterface` is straightforward as they all have similar semantics. For this reason the transformation only requires to swap instructions and customise descriptor according to the table. See Listings 4 and 5 for an example presenting simplified byte-code instructions. These instructions may all be used in cross-component invocations and thus each occurrence is a candidate for replacement by the bytecode enhancer.

The situation is less obvious for `invokespecial`. In this case, the instruction has several usages, including the invocation of private methods, the invocation of methods via the `super` keyword, and the invocation of constructor using the `this` keyword [26, sect 6.5]. We support the transformation of `invokespecial` used with `super` and constructors [6]. For `invokespecial (super)`, the transformation is equivalent to the transformations applied to `invokevirtual` and `invokeinterface`.

The transformation of `invokespecial` used in conjunction with the `new` keyword at object allocation sites is less straightforward. In bytecode, the respective methods invoked all have a special name `<init>`. A correct transformation requires the detection of a usage pattern that consists of a certain sequence of instructions, the so-called *bytecode behaviour*. The pattern for constructors consists of the following three instructions (1) `new C` (2) `dup` (3) `invokespecial C.<init>:(A*)V` [26, sect 5.4.3.5], possibly with some intermediate instructions. The semantics of this sequence is: (1) create a new object of type C and push it onto the stack, (2) duplicate the object on the top of the stack and (3) invoke the constructor. The object on the top of the stack is duplicated because one element is consumed (popped) by the constructor invocation and the object must remain on top of the stack so that the value can be assigned to a variable (usually using an `astore` instruction). To

[6] Invocations of private methods are out of scope as they can not be used for cross-component invocations.

```
1 new lib/Foo
2 dup
3 bipush 10
4 invokespecial lib/Foo."<init>":(I)V
```

Listing 6 Bytecode for `new lib.Foo(10)`.

```
1 bipush 10
2 invokedynamic C$D:(I)Llib/Foo;
```

Listing 7 Enhanced bytecode for code in listing 6.

```
1 java -cp dynamo<..>.jar org.dynamo.compiler.Compiler -sourcepath ./src -d ./bin
```

Listing 8 Basic use of dynamo to compile all Java source code files in the `src` folder and store the .class files in the `bin` folder.

replace this sequence, it must be changed to a standard method invocation, which means that the instructions `new C` and `dup` must be dropped and `invokespecial` must be replaced by `invokedynamic`. Note that `invokedynamic` returns a value, this value is pushed onto the stack and effectively replaces the two dropped instructions. If the constructor contains parameters, the transformed sequence will retain the instructions to push these parameters onto the stack before the `invokedynamic` instruction is executed. An example transformation is shown in Listings 6 and 7.

Since `invokedynamic` cannot use `<init>` as the name parameter for the bootstrap method [26, sect 4.10], an artificial name set to `C$D`, meaning "constructor dynamic", is used instead. The use of the $ sign in method names is legal, but according to the language specification "The $ sign should be used only in mechanically generated source code or, rarely, to access pre-existing names on legacy systems" [22, sect 3.8]. By complying with this rule it is very unlikely that this name choice will create conflicts with user code.

The actual bootstrap methods referenced in the modified bytecode are defined as static methods in the class `com.dynamo.rt.DynamoBootstrap`. There is one method for each instruction type (`bootstrapInterface`, `bootstrapStatic`, etc). The semantics of these methods is described in section 5.

4.2 The Dynamo Compiler

The dynamo compiler is implemented as a post compiler. It is a wrapper around the dynamo enhancer, combined with the standard Java compiler accessed via the compiler API (JSR-199) [1]. More explicitly, the dynamo compiler first uses the standard Java compiler through JSR-199 to compile compilation units in memory. The resulting bytecode is represented using a map that associates fully qualified class names with byte arrays. The compiler then uses the enhancer to apply bytecode transformations using information from (1) the compiler runtime parameters specifying the classpath, (2) compiler runtime parameters specifying filters.

The classpath information is used to determine component boundaries. The filters are described in the next section, the options of the compiler command line interface (CLI) are described in appendix B.

Figure 1 shows the design of the dynamo compiler; Listing 8 shows the basic usage of the compiler via its CLI.

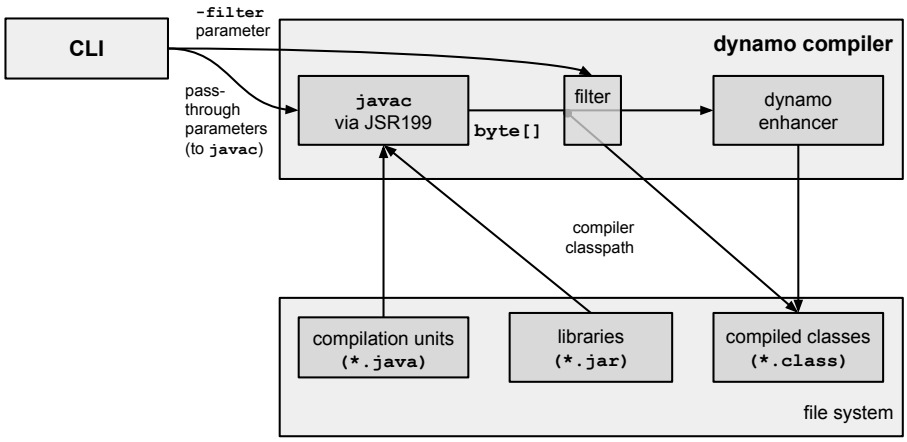


Figure 1 Compiler Design.

4.3 The Dynamo Filter DSL

It is often desirable for users to retain fine-grained control over the compilation process. One of the more common use cases is not to use `invokedynamic` when invoking functionality from Java platform libraries, due to the strong emphasis on backward compatibility in these libraries [8, 9].

For this reason, we have developed a lightweight domain-specific language that can be used to filter the methods and call sites where `invokedynamic` is used. These filters are used to select *invocation records* consisting of two methods, the method containing the call site, and the target method invoked before enhancement takes place.

The method filter is composed of individual filters that all have the following properties:

1. a *kind* (`+`, `-`) defining whether a filter is an include or an exclude filter
2. a *role* (`callsite`, `target`) defining whether a filter applies to the client method that has the call site, or to the target method to be invoked
3. a *class pattern* defining the classes to which the filter applies. Class names are fully qualified, with a dot used as package separator
4. an optional *method name pattern* defining the methods to which the filter applies
5. an optional *descriptor pattern* defining the descriptors to which the filter applies. This allows for the discrimination of overloaded methods. Descriptor patterns use the syntax defined in the JVM specification [26, sect 4.3.3] plus optional wild card characters.

Filters can be defined using a simple domain specific language; the grammar of this language is given in appendix A. The `*` and `?` wild cards can be used in class names, method names and descriptor patterns. The *default filter* is defined using the patterns listed in Table 2.

The intention of this filter is to exclude all invocations of target methods defined in the Java Development Kit. A user-defined filter can modify the default filter by using additional exclude and include patterns. The main use case for include patterns is to selectively permit the enhancement of JDK method invocations. With additional exclude patterns, users can manually specify that enhancements for certain targets are not required since the libraries in which the respective methods are defined are known to be API stable. In case additional include and exclude filters conflict, we resolve this as follows: invocation records are accepted if they are accepted by at least one of the include filter and not accepted by any of the exclude filters. This resolution is consistent with how include and exclude patterns are handled in

Table 2 The default filter.

Filter definition	Description
`+callsite *,+target *`	include all invocation records

`-target java.*`
`-target javax.*`
`-target com.sun.*`
`-target sun.*`
`-target com.oracle.*`
`-target org.ietf.*`
`-target org.omg.*`
`-target org.w3c.*`
`-target org.xml.*`
	then exclude all packages included in the Java development kit.

Table 3 Filter examples.

Filter definition	Description
`-callsite com.foo.Bar`	exclude invocations from call sites within `com.foo.Bar`
`-target com.foo.*`	exclude invocation of targets in methods in packages starting with `com.foo`
`+target java.lang.String#substring`	include invocations of the `substring` methods defined in `String`
`+target java.lang.String#substring(I)*`	include invocations of the `substring(int)` method defined in `String`

popular build tools, and we assume that developers are familiar with the practice. Some examples of filters are given in Table 3.

The definition of the default filter is relatively coarse. For instance, the filter would also include a package with the prefix `com.oracle` that is not part of the Java developer kit, such as certain JDBC drivers. In order to override this behaviour, users must use custom filters.

5 Linking

In this section, we describe the bootstrap process. We first discuss the strategy we are using to locate and select the target method. In a nutshell, we use an algorithm that aims at aligning linking behaviour with compile time behaviour to facilitate program comprehension by developers. We then describe the complexity of the algorithm, and some implementation issues.

5.1 Resolution

At runtime, we need to resolve the reference in the bootstrap method and locate an actual target method. This method must be *adaptable* to the original method. We only look for methods that are non-abstract, visible from the call site and have the same name and arity, but may have a different descriptor or are defined in a super type. This is similar to the problem the compiler has to solve when it selects the most specific method [22, sec 15.12.2.5]. However, we also have to take the return type into account. Not only can the return type change as we allow specialisation or narrowing of the return type, but we potentially also

have to deal with return type overloading, a situation that can only arise in bytecode. In particular, the standard Java compiler uses synthetic [26, sec 4.7.8] bridge methods to deal with co-variant return types. This means that any runtime method resolution has to be able to deal with a situation where there are multiple methods with the same name, the same parameter types but different return types within one class. Since return type overloading is not supported by the Java language, we can make the assumption that if this situation arises, only one of these methods is non-synthetic.

We refer to the selection of an adaptable method as *runtime resolution*, or in this context just *resolution* for short. Resolution is defined with respect to adaptations that can be performed by method handles [35]. Different resolution strategies are possible: (1) a *greedy strategy* that locates any adaptable method and selects it, (2) an *optimal strategy* that tries to find the "best match" according to some metric and (3) an *unambiguous best match strategy* that tries to find a method that is not only the best match in the sense of being optimal with respect to some order, but must also be strictly better than other candidates with respect to this order.

As one of our objectives is to address inconsistencies between source and binary compatibility, we decided to use strategy (3) to mimic the compiler behaviour of choosing the most specific method [22, sect 15.12.2.5]. While the overall aim is somehow similar to how the compiler processes method invocation expressions [22, sect 15.12], there are important differences caused by the differences in representation of language features such as generic types, static imports and varargs in source code and bytecode.

For a concise formulation of the algorithm, we propose a simple model, the *type conversion graph (TCG)*. This graph captures subtype relationships between classes and interfaces, boxing and unboxing relationships between wrapper types and their respective primitive types, and widening conversion relationships between primitive types [26, sect 5.1.2]. We build the TCG for a program by applying the following rules:

1. All (primitive and reference) types except annotation types but including array types that occur in the program are added as vertices to TCG.
2. If T_1 and T_2 are (non-generic) class, interface or array types and T_1 is a direct subtype of T_2 as defined in [22, sects 4.10.2, 4.10.3] then an edge $T_1 \to T_2$ with a label r is added to TCG.
3. If T_1 and T_2 are primitive types and there is a widening conversion from T_1 to T_2 as defined in [22, sect 5.1.2] then an edge $T_1 \to T_2$ with a label p is added to TCG.
4. If T is a primitive type, and Cl is the wrapper type of T, then an edge $T \to Cl$ with a label b (for boxing) and an edge $Cl \to T$ with a label u (for unboxing) are added to TCG.

Figure 2 shows an example TCG. The intention behind the TCG is that paths between types represent valid (potentially composite) conversions. However, the compiler imposes additional constraints. For instance, a widening primitive conversion followed by a boxing conversion is not permitted in assignment and invocation contexts [22, sects 5.2, 5.3]. As our motivation is to align compilation and linking behaviour, we impose the same restrictions. We do so by defining a *valid path* between two types considered as vertices in the TCG as a path such that the labels of the edges in this graph spell a word defined by the following simple regular grammar: `p|r*|br*|up`. This grammar is derived from the rules used in [22, sects 5.2, 5.3][7].

[7] The identity conversion is represented by a path of length 0. The primitive widening conversion rules in [22, sect 5.1.2] use the transitive closure for primitive widening conversions, we can therefore use `p` instead of `p*` in the grammar

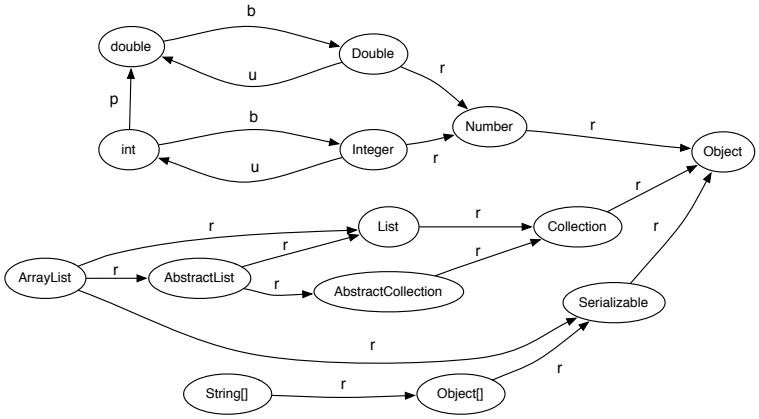


Figure 2 Example Type Conversion Graph.

When we try to locate an adaptable method, we only consider methods that are non-synthetic[8]. Given the TCG, the adaptability relationship between types $\sqsubseteq_{type} \subseteq T \times T$ can be easily defined as $t_1 \sqsubseteq_{type} t_2$ iff there is a valid path from t_1 to t_2 in the TCG. In order to identify, compare and select adaptable methods, we need to analyse their (1) defining type (2) parameter types and (3) return types. We define an order between methods based on their *extended descriptor* (*XD*) comprising the type where the method is defined, followed by the parameter types and the return type. We use the syntax `<defining_type>(parameter_type*)return_type` for XDs.

The relation $\sqsubseteq_{type}$ can be easily promoted to a relationship between XDs of the same arity: $T^1(A_1^1,..,A_n^1)R^1 \sqsubseteq_{desc} T^2(A_1^2,..,A_n^2)R^2$ iff $R^2 \sqsubseteq_{type} R^1$ and $T^1 \sqsubseteq_{type} T^2$ and $A_i^1 \sqsubseteq_{type} A_i^2$ for each i. Note that the direction of $\sqsubseteq_{type}$ is reversed ("contravariant") for return types. By treating the defining type as a virtual first parameter, we include possible target methods defined in super types. For instance, this allows adaptations in cases where methods have been pushed up the type hierarchy by refactoring[9].

We can then define a simple disambiguation algorithm as follows: given an extended descriptor $\Delta = T(A_1,..,A_n)R$ and a set of adaptable descriptors $\{\Delta^i\}$, where $\Delta \sqsubseteq_{desc} \Delta^i$, we chose a descriptor Δ^k from $\{\Delta^i\}$ if $\Delta^k \sqsubseteq_{desc} \Delta^i$ for all $i \neq k$. This captures the intention of selecting the *most specific method* that is adaptable. In case there is more than one minimal XD with respect to $\sqsubseteq_{desc}$, and the respective methods have the same parameter types, we chose the method with the more specific return type following the compiler [22, sect 15.12.2.5]. If resolution fails to detect a unique XD, the process will result in a linkage error (instance of `java.lang.NoSuchMethodError`).

An example where this occurs is when a method `Object foo(java.util.ArrayList)` is replaced by two methods `String foo(java.util.AbstractList)` and `Object foo(java.io.Serializable)`[10] within the same class. Both methods are suitable targets, but disambiguation fails to detect a best method that is strictly better than the other one.

A side effect of the resolution algorithm just described is that the only possible targets for replaced constructor invocations are constructors defined within the same class. This

[8] This will remove ambiguity if several methods with the same name and parameter type but different return types are present. This is discussed in more details in section 7.2

[9] The standard JVM runtime method resolution already includes superclass lookup [26, sect 5.4.3.3]

[10] The class `ArrayList` extends `AbstractList` and implements the interface `Serializable`.

follows from the fact that we only look for targets with the same name as the original target (i.e., `<init>`). While we also consider constructors from super classes as possible targets, these methods have a more general return type, and therefore do not qualify. It follows further that the type returned by the `invokedynamic` invocation is the same as the type of the original (`invokespecial (new)`) invocation.

5.2 Algorithm Complexity

Adaptability is defined with respect to reachability in the TCG. Reachability is a potentially expensive operation. The worst case complexity of standard shortest path algorithms used to query adaptability between two types is quadratic [15], while the pre-computation of a reachability index that enables constant time queries is super-quadratic but sub-cubic [4]. However, the paths that need to be traversed by the algorithm are generally short because of the flat hierarchies found in most Java programs [39], and the complexity of standard shortest path and many reachability algorithms is near linear for sparse graphs.

We provide further evidence of the performance of our approach in the evaluation section.

5.3 Implementation Issues

The runtime component is implemented as a small library that must be included in the class path of applications compiled or enhanced with dynamo. The method lookup algorithm is implemented in `org.dynamo.rt.DynamoBootstrap`. This class has five static bootstrap methods corresponding to the different instructions (`bootstrapVirtual`, `bootstrapStatic`, `bootstrapInterface`) and the two variants of `invokespecial` supported (`bootstrapSpecialNew`, `bootstrapSpecialSuper`). These methods differ slightly in terms of decoding and interpreting method types according to Table 1. They all invoke a common method which finds the most suitable method as described in section 5.1.

We use reflection to gather information about types, their relationships and members, and to reason about this information. Although reflection is not ideal as it may trigger some unnecessary class loading if classes are analysed for members that are not used at the end, it facilitates the implementation of bootstrap methods. The actual target finder algorithm produces an instance of `java.lang.reflect.Method` which is easily converted to a method handle using the `MethodHandles.Lookup.unreflect()` protocol and its variants for constructors and special methods. The actual conversion of types is then easily performed by `MethodHandle.asType()`. At the end of this process, a constant call site wrapping this handle is instantiated.

The method `MethodHandle.asType()` supports all of the type conversions we use and therefore effectively performs the transformation of both return and parameter types. Since we replace classic invocations with `invokedynamic`, we automatically support cases where interfaces are converted to classes or vice-versa. Finally, changes where non-static target methods have evolved to static methods are handled by ignoring the first parameter when the method is invoked. This way, the descriptor described in Table 1, row 2 is produced. The transformation is achieved by using the `MethodHandles.dropArguments` API. The (simplified) implementation of a bootstrap method is shown in Listing 9.

6 Benchmarks

In this section we discuss sets of benchmarks used to test various dynamo components, and to assess the performance overhead induced by dynamo.

```
import java.lang.invoke.*;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;

...
CallSite bootstrapVirtual(Lookup caller, String name, MethodType type) {
   Class<?> owner = type.parameterType(0);
   Method method = find(owner, name, type); // use resolution with TGC
   MethodHandle handle = lookup.unreflect(method);
   if (Modifier.isStatic(method.getModifiers())) {
      handle = MethodHandles.dropArguments(handle, 0, method.getDeclaringClass());
   }
   return new ConstantCallSite(handle.asType(type));
}
...
```

Listing 9 Bootstrap method implementation example (simplified).

6.1 Compiler Benchmarks

The compiler benchmarks are based on a comprehensive set of unit tests we have developed to quality assure dynamo. Tests are based on scenarios, each scenario consists of three classes:

1. `version1/lib/Foo.java` - the source code of version 1 of a class `lib.Foo` providing a method
2. `version2/lib/Foo.java` - the source code of version 2 of a class `lib.Foo` providing a modified method
3. `client/Main.java` - the source code of the client program using the method provided by `Foo`

During testing, the following sequence is executed for each scenario:

- Compile both versions of `Foo` and package them in different libraries (`lib-v1` and `lib-v2`). This should succeed.
- Compile `Main` with javac against `lib-v1`. This should succeed.
- Run the compiled class `Main` with `lib-v1`. This should succeed.
- Run the compiled class `Main` with `lib-v2`. This should fail.
- Recompile `Main` with dynamo against `lib-v1`. This should succeed.
- Run the recompiled class `Main` with `lib-v2`. This should succeed, except for the last two "ambiguous" benchmarks designed to produce linkage errors.

Scenarios are identified by self-descriptive unique names. For instance `invokeinterface_narrow_ret` is a scenario that contains an `invokeinterface` instruction with a reference to a method that evolves by narrowing the return type. Table 4 shows an overview of the scenarios used for testing, the first column describes the change pattern, the other columns show the type of invocation that is being converted. These scenarios were designed to obtain full coverage of all possible combinations for common scenarios, and good partial coverage for more exotic cases such as chained conversions (like boxing and then generalising a parameter type). A check mark in the row-column intersection means that there is such a scenario, `n/a` indicates that such a scenario is impossible (example: return type conversions for constructor invocations) or does not make sense (example: converting an invocation of a certain type to an invocation of the same type without adapting parameter or return types).

Table 4 Benchmark overview.

	`invoke-static`	`invoke-virtual`	`invoke-interface`	`invoke-special (super)`	`invoke-special (new)`
narrow return type	✓	✓	✓	✓	n/a
specialised return type	✓	✓	✓	✓	n/a
specialised array return type	✓	✓	✓	✓	n/a
box return type	✓	✓	✓	✓	n/a
unbox return type	✓	✓	✓	✓	n/a
widen parameter type	✓	✓	✓	✓	✓
generalise parameter type	✓	✓	✓	✓	✓
generalise array parameter type	✓	✓	✓		
generalise array parameter to `Object`			✓		
generalise array parameter to `Serializable`			✓		
generalise array parameter to `Cloneable`			✓		
box parameter type	✓	✓	✓	✓	✓
unbox parameter type	✓	✓	✓	✓	✓
multiple methods w/ generalised parameters	✓				
convert to `invokestatic`	n/a	✓	✓	n/a	n/a
convert to `invokeinterface`	n/a	✓	n/a	n/a	n/a
convert to `invokevirtual`	n/a	n/a	✓	n/a	n/a
unbox and widen parameter type	✓	✓	✓		✓
box and narrow return type	✓	✓	✓		n/a
box and generalised parameter type	✓	✓	✓	✓	✓
method ambiguous parameter		✓			
method ambiguous two parameters		✓			

We use the test scenarios as compiler micro-benchmarks. However, as we do not test linking at this stage, for some scenarios the code compiled with dynamo (`Main.java`) is identical. For instance, this is the case for the various "generalise array parameter .. " scenarios. We remove these scenarios from the set of 65 scenarios listed in Table 4 in order to avoid double counting, the result is a set of 49 scenarios we can use for benchmarking.

Performance measurement in Java is not straightforward as Java uses runtime optimisation (JIT, HotSpot). To account for this, repeated invocations and JVM warm-up runs are necessary to produce statistically meaningful results [20]. For this reason, we used JMH,[11] a tool that provides the respective features. For each experiment, we executed 15 warm-up and 30 trial runs. For the experiments, we used the Java(TM) SE Runtime Environment (build 1.8.0_20-b26) with a Java HotSpot(TM) 64-Bit Server VM (build 25.20-b23, mixed mode) on a MacBook with OSX 10.5.5, an 2.8 GHz Intel Core i7 processor, 16 GB 1600 MHz DDR3 and SSD disk.

Table 5 summarises the performance results. We compare the dynamo compiler with the standard Java compiler accessed through JSR199. To achieve a fair comparison, we compile in memory in both cases and try to measure the net effect of the post compilation step when bytecode is manipulated. We report average, standard deviation and confidence intervals, all in ms. The fourth column contains the number of benchmarks tested, and column 5 is the average divided by this number, i.e., the typical time a single compilation takes. The results

[11] `http://openjdk.java.net/projects/code-tools/jmh/`

Table 5 Dynamo vs classic compiler performance.

benchmarks	average runtime (ms)	stdev (ms)	bench-mark count	average runtime single benchm. (ms)	confidence interval (ms) (99.9%)
classic	1923	65	49	39.24	[1857, 1988]
dynamo	2142	91	49	43.71	[2051, 2233]

Table 6 Method resolution performance.

benchmarks	average runtime (ms)	stdev (ms)	bench-mark count	average runtime single benchm. (ms)	confidence interval (ms) (99.9%)
all	0.180	0.001	14	0.013	[0.179, 0.181]

clearly show that the compile time overhead imposed by dynamo is very small.

6.2 Runtime Benchmarks

We have also created a benchmark for runtime resolution. As the low performance overhead of `invokedynamic` is known and has been reported in previous research [29, 3], the benchmark is based on unit tests that only compute the target methods using the resolution algorithm described in 5.1 from a set of candidate methods. Therefore, the design of the benchmarks is driven by the following considerations: (1) a benchmark has a (significant) number of candidate target methods scattered across several classes within the type hierarchy, (2) candidate methods are detected via conversions represented by non-trivial paths within the TCG including combinations of different edge types.

The individual benchmark scenarios are implemented as standard JUnit tests, and used for quality assurance as well as to assess performance. Each test case has a self-explanatory name to express its purpose. The packages and respective test cases are:

- `specstaticreturntype` – a simple scenario with a test to locate a static target method with a specialised return type
- `boxunbox` – a scenario with tests that require the following conversions to locate the target method: (1) boxing, (2) un-boxing, (3) boxing and widening of parameter types and (4) un-boxing and narrowing of the return type
- `methodoverloading` – a scenario with multiple d potential target methods, and tests checking different resolution strategies to select target methods
- `overloadinghierarchy` – a scenario with multiple overloaded potential target methods scattered across a class and its direct and indirect super classes
- `disambiguatebyreturntype` – a scenario to test situations where the target method is selected based on the return type, simulating a scenario similar to what will be further discussed in section 7.2

Table 6 summarises the performance results, using the same format as table 5 described above. This result confirms that the performance overhead of runtime resolution is negligible.

7 Case Studies

To demonstrate the applicability of our approach, we present two case studies sourced from real world scenarios.

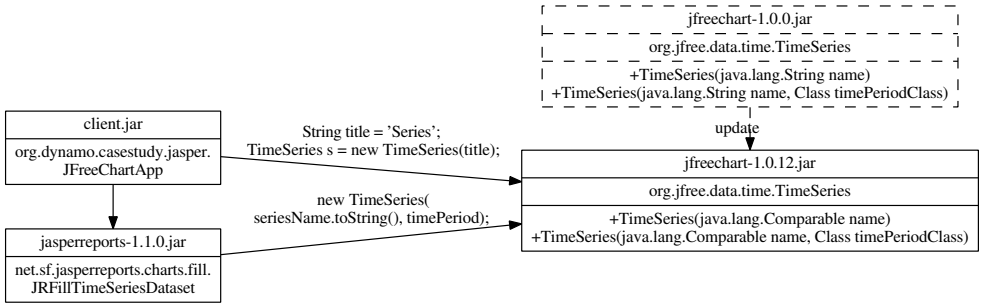


Figure 3 Case study 1 application design.

7.1 Solving Compatibility Issues between Jasperreports and Jfreechart

The first case study is sourced from a real world scenario using the popular Java reporting library *jasperreports*[12]. *Jasperreports* uses several other libraries including *jfreechart*[13], which evolve independently, and this can lead to incompatibilities. In particular, a problem occurs when *jfreechart* evolves from version 1.0.0 to 1.0.12. In this upgrade, the constructor `TimeSeries(String)` in the class `org.jfree.data.time.TimeSeries` is changed to `TimeSeries(Comparable)`. There are several overloaded variants of this constructor, but for all of them the type of the first parameter is changed from `String` to `Comparable`.

This change is particularly dangerous for two reasons: (1) The change happens between two micro versions, such evolutions are widely regarded as being compatible in accordance with popular semantic versioning schemes [30, 31]. These schemes are widely used to represent compatibility contracts between collaborating components, and tools and frameworks like Maven and OSGi support them. (2) This change only affects binary compatibility, and its implications therefore depend on the mode of deployment. The issue does not occur when *jasperreports* is built, but only if a dynamic system upgrade mechanism like OSGi or Java WebStart is used. The key observation here is that `String` implements the `Comparable` interface, and that generalising a parameter type generally preserves source compatibility.[14]

The system used in the case study is depicted in Figure 3. The system is represented using a model resembling a UML class diagram. However, the arrows represent the invocation of a method or constructor in the target class, and the dashed box represents the older version of the library. Labels on edges show the Java code at the respective call site. The model contains an application `client.jar`, which is a simple client application producing a basic report containing a chart. The example illustrates both compile and runtime dependencies. While the client application is compiled and invoked against both libraries, *jasperreports* runs only against *jfreechart-1.0.0* and this dependency is not resolved at compile time.

This setting allows for at least three scenarios: (1) the client is compiled and executed with *jfreechart-1.0.0*, (2) the client is compiled against *jfreechart-1.0.0* and executed with *jfreechart-1.0.12*, (3) the client is compiled and executed with *jfreechart-1.0.12*. Only the first scenario succeeds with the standard Java compiler. In the second scenario, both dependencies *client* → *jfreechart* and *jasperreports* → *jfreechart* will fail due to the incompatible change in `TimeSeries(..)`. In the third scenario, the problem will be solved for the client as it can

[12] `http://community.jaspersoft.com/project/jasperreports-library`
[13] `http://www.jfree.org/jfreechart/`
[14] There are exceptions where generalising a parameter type can create ambiguities in the presence of overloading that lead to compilation errors.

```
ldc           #10  // String 'Series'
astore_2
new           #11  // class org/jfree/data/time/TimeSeries
dup
aload_2
invokespecial #12 // Method org/.../TimeSeries."<init>":(Ljava/lang/String;)V
```

Listing 10 Client code compiled with javac.

```
ldc           #10     // String 'Series'
astore_2
aload_2
invokedynamic #130, 0 // InvokeDynamic #3:C$D:(Ljava/../String;)Lorg/../TimeSeries;

BootstrapMethods:
 3: #126 invokestatic org/dynamo/rt/DynamoBootstrap.bootstrapSpecialNew:(
          Ljava/lang/invoke/MethodHandles$Lookup;
          Ljava/lang/String;
          Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;
```

Listing 11 Client code compiled with dynamo.

now be compiled, although invocations of `TimeSeries(..)` from *jasperreports* will still fail. However, the working client will remain prone to similar problems as *jfreechart* may evolve further.

To solve this problem, we compiled the client with the dynamo compiler. This produces a client program that is compatible with both versions of *jfreechart* and resilient to any possible future change of a similar nature. We also processed the *jasperreports* library with the dynamo enhancer, demonstrating the application to third party or legacy software where only bytecode is available. Finally, we invoked the program and verified by inspection that both versions produce the same report. Listings 10 and 11 show the bytecode of the original client compiled with the standard Java compiler followed by the enhanced bytecode. The output is provided in the format produced by the `javap` tool, with comments revealing values defined in the constant pool. The modifications follow the process described above: the instructions creating the object and invoking its constructor are replaced by `invokedynamic`, which refers to the respective bootstrap method.

We have also used this case study as a macro benchmark in order to measure the overhead of using dynamo on a real-world program. Again, JMH was used to conduct these experiments, with 15 warm-up and 30 trial runs. Table 7 summarises the compiler results, and confirms that the performance overhead of using dynamo is small.

We have also measured the runtime overhead. In this benchmark, a simple report is generated first by running a simple demo program that uses *jasperreports* compiled with the classic compiler, and then by running it again with *jasperreports* compiled by dynamo. We do use the same version of *jfreechart* (1.0.0) in both cases in order to avoid a bias that would be caused by the different performance characteristics of different versions of this library. Therefore, the same classes are used in both cases, but the generated bytecode and the method used for linking differs. The respective results are reported in table 8. This again confirms that the overhead of dynamo runtime resolution is small.

Table 7 Dynamo vs classic compiler performance for *jasperreports*.

benchmark	average runtime (ms)	stdev (ms)	confidence interval (ms) (99.9%)
classic	1278	52	[1255,1302]
dynamo	1330	56	[1305,1355]

Table 8 Runtime performance of simple report generation with *jasperreports*.

benchmark	average runtime (ms)	stdev (ms)	confidence interval (ms) (99.9%)
classic	168	9	[165,172]
dynamo	175	12	[154,218]

7.2 Avoiding the Hazards of Covariant Return Types and Bridge Methods

In a second case study, we demonstrate how compilation with dynamo can avoid an error that was encountered when the JDK was refactored to use more covariant return types when overriding `clone()` methods[15]. We use the formulation of this problem presented in [34]. The scenario consists of three simple classes, `Wrapper`, `WrapperChild` and `WrapperGrandchild`, the source code is shown in listing 12. Both `WrapperChild` and `WrapperGrandchild` override `wrap`, but `WrapperGrandchild` does so using a covariant return type. When a client calls `wrap` on an object that is an instance of `WrapperGrandchild` but declared as `Wrapper`, an `invokevirtual` instruction pointing to a `wrap` method with the descriptor `(LObject;)LCollection;` (package names omitted) is used. Therefore, a method with such a descriptor must exist in `WrapperGrandchild`. The compiler solves this problem by generating a synthetic bridge method with the descriptor `(LObject;)LCollection;` that then just delegates to the actual method with the descriptor `(LObject;)LList;`. This is therefore a case of return type overloading. The respective bytecode is shown in listing 13[16].

The problem occurs when `Wrapper` and `WrapperChild` on the one hand and `WrapperGrandchild` on the other hand are deployed in different libraries. When `WrapperChild` is refactored to also use a covariant return type, it too gets two `wrap` methods - the actual method with the descriptor `(LObject;)LList;` and the synthetic methods with the descriptor `(LObject;)LCollection;`. Now assume we execute the code in listing 14.

Executing this line invokes `WrapperGrandchild.wrap:(LObject;)LList;`. This method calls `WrapperChild.wrap:(LObject;)LCollection;` via `super` as this was the only method in `WrapperChild` that was available when `WrapperGrandchild` was compiled against the old version of `WrapperChild`. This method is now a bridge method, and therefore calls `WrapperChild.wrap:(LObject;)LList` using an `invokevirtual` instruction. Since the actual type of the receiver object is `WrapperGrandchild`, this call is dispatched to `WrapperGrandchild.wrap:(LObject;)LList;`. This causes the program to loop infinitely and terminate with a `StackOverflowError`. The respective call graph is shown in Figure 4.

Using the dynamo compiler prevents this from happening not because of the signature adaptation, but due to the disambiguation strategy used during runtime resolution. Resolution only considers non-synthetic methods. Therefore, we have two ad-

[15] `http://mail.openjdk.java.net/pipermail/core-libs-dev/2012-January/009119.html` [Accessed: October 20, 2015]

[16] The repeated `checkcast` instruction is a Java compiler bug reported in `http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6246854`[Accessed: May 3, 2016]

```
public class Wrapper {
    public Collection wrap(Object o) {
          Collection c = new ArrayList();
          c.add(o);
          return c;
    }
}

public class WrapperChild extends Wrapper {
    @Override public Collection wrap(Object o) {
          return super.wrap(o);
    }
}

public class WrapperGrandchild extends WrapperChild {
    @Override public List wrap(Object o) {
          return (List) (super.wrap(o));
    }
}
```

Listing 12 Case study 2 source code (imports omitted).

```
// access flags 0x1
public wrap(Ljava/lang/Object;)Ljava/util/List;
ALOAD 0
ALOAD 1
INVOKESPECIAL WrapperChild.wrap (Ljava/lang/Object;)Ljava/util/Collection;
CHECKCAST java/util/List
CHECKCAST java/util/List
ARETURN

// access flags 0x1041
public synthetic bridge wrap(Ljava/lang/Object;)Ljava/util/Collection;
ALOAD 0
ALOAD 1
INVOKEVIRTUAL WrapperGrandchild.wrap (Ljava/lang/Object;)Ljava/util/List;
ARETURN
```

Listing 13 Case study 2 bytecode of `WrapperGrandchild`.

```
new WrapperGrandchild().wrap("x");
```

Listing 14 Case study 2 client code.

```
1  public wrap(Ljava/lang/Object;)Ljava/util/List;
2   aload_0
3   aload_1
4   invokedynamic #31 // wrap:(LWrapperChild;Ljava/lang/Object;)Ljava/util/Collection;
5   checkcast java/util/List
6   checkcast java/util/List
7   areturn
8
9  public synthetic bridge wrap(Ljava/lang/Object;)Ljava/util/Collection;
10  aload_0
11  aload_1
12  invokevirtual #4 // Method wrap:(Ljava/lang/Object;)Ljava/util/List;
13  areturn
14
15 Constant pool:
16  #4 = Methodref // WrapperGrandchild.wrap:(Ljava/lang/Object;)Ljava/util/List;
17 #31 = InvokeDynamic// wrap:(LWrapperChild;Ljava/lang/Object;)Ljava/util/Collection;
```

Listing 15 Bytecode of `WrapperGrandchild` generated by dynamo.

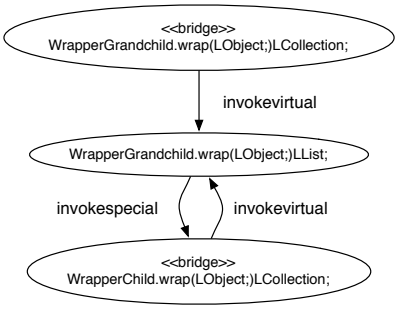


Figure 4 Call graph after refactoring, compiled with javac.

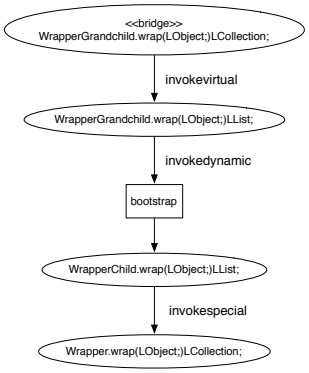


Figure 5 Call graph after refactoring, compiled with dynamo.

aptable candidate methods with the extended descriptors `Wrapper:(Object)Collection` and `WrapperChild:(Object)List`. Both are minimal (most specific), but because they both have the same parameter types, the method with the more specific return type `WrapperChild:(Object)List` is selected. The replacement pattern used is `invokespecial` (`super`) as the call via `super` is the cross-component call. The bytecode produced by *dynamo* is shown in Listing 15, the respective call graph is shown in Figure 5.

8 Conclusion

In this paper, we have proposed the dynamo framework consisting of a bytecode enhancer, a compiler, a filter DSL and a runtime component. We have demonstrated the benefits of using dynamo using a set of benchmarks and two case studies sourced from real-world scenarios. In both case studies, unintuitive runtime errors can be avoided by using the dynamo compiler. The benchmarks show that the overhead of using dynamo is low.

While we have discussed dynamo in the context of the Java language and compilation targeting the JVM, the same idea can be potentially applied to other languages and platforms.

The enhancer can be used as a tool kit to improve other compilers targeting the JVM, such as `scalac`. Dynamo could also be ported to other language / platform combinations facing similar problems related to binary compatibility, such as `C#/CLR/.NET`.

In the context of Java, the same idea could also be applied to support compatible type changes for fields exposed by libraries. We did not include this in our work as direct field access in Java programs is discouraged. Other possible extensions include to restrict constant inlining across components.

Acknowledgements. The authors would like to thank Alex Buckley, Nicholas Hollingum, Stepanka Jezkova and Alex Potanin for their valuable feedback.

References

1. JSR-000199 Java™ Compiler API. `https://jcp.org/aboutJava/communityprocess/final/jsr199/index.html`. [Accessed: October 20, 2015], 2006.
2. Miles Barr and Susan Eisenbach. Safe upgrading without restarting. In *Proceedings ICSM'03*, pages 129–137, 2003.
3. Patricia Conde and Francisco Ortin. JINDY: A java library to support invokedynamic. *Computer Science and Information Systems*, 11(1):47–68, 2014.
4. Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings STOC'87*. ACM, 1987.
5. John Corwin, David F. Bacon, David Grove, and Chet Murthy. Mj: a rational module system for java and its applications. In *Proceedings OOPSLA'03*. ACM, 2003.
6. Bradley E Cossette and Robert J Walker. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *Proceedings FSE'12*. ACM, 2012.
7. Ilie Şavga and Michael Rudolf. Refactoring-based support for binary compatibility in evolving frameworks. In *Proceedings GPCE '07*. ACM, 2007.
8. Joseph D. Darcy. Kinds of compatibility: Source, binary, and behavioral. `https://blogs.oracle.com/darcy/entry/kinds_of_compatibility`. [Accessed: October 20, 2015], 2008.
9. Joseph D. Darcy. JDK release types and compatibility regions. `https://blogs.oracle.com/darcy/entry/release_types_compatibility_regions`. [Accessed: October 20, 2015], 2009.
10. Jim des Rivières. Evolving Java-based APIs. `http://wiki.eclipse.org/Evolving_Java-based_APIs`. [Accessed: Nov. 20, 2015], 2007.
11. Jens Dietrich, Kamil Jezek, and Premek Brada. Broken promises - an empirical study into evolution problems in java programs caused by library upgrades. In *Proceedings CSMR-WCRE'14*. IEEE, 2014.
12. Jens Dietrich, Kamil Jezek, and Premek Brada. What java developers know about compatibility, and why this matters. *Empirical Software Engineering*, pages 1–26, 2015.
13. Danny Dig and Ralph Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, 2006.
14. Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph Johnson. ReBA: refactoring-aware binary adaptation of evolving libraries. In *Proceedings ICSE'08*. ACM, 2008.
15. Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.
16. Mikhail Dmitriev. Language-specific make technology for the Java programming language. In *Proceedings OOPSLA '02*, pages 373–385, New York, NY, USA, 2002. ACM.
17. Sophia Drossopoulou, David Wragg, and Susan Eisenbach. What is Java binary compatibility? In *Proceedings OOPSLA'98*. ACM, 1998.

18 David Flanagan and Yukihiro Matsumoto. *The ruby programming language*. O'Reilly Media, Inc., 2008.

19 Ira R. Forman, Michael H. Conner, Scott H. Danforth, and Larry K. Raper. Release-to-release binary compatibility in SOM. In *Proceedings OOPSLA '95*. ACM, 1995.

20 Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings OOPSLA'07*. ACM, 2007.

21 Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

22 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java™ Language Specification (Java SE 8 Edition)*. Oracle America, Inc., February 2015.

23 Chanwit Kaewkasi. Towards performance measurements for the java virtual machine's invokedynamic. In *Proceedings VIML'10*. ACM, 2010.

24 Ralph Keller and Urs Hölzle. Binary Component Adaptation. In *Proceedings ECOOP '98*. Springer, 1998.

25 M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

26 Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java™ Virtual Machine Specification (Java SE 8 Edition)*. Oracle America, Inc., February 2015.

27 Tom Mens, Juan Fernández-Ramil, and Sylvain Degrandsart. The Evolution of Eclipse. In *Proceedings ICSM'08*. IEEE, 2008.

28 Charles Nutter. A first taste of invokedynamic. `http://blog.headius.com/2008/09/first-taste-of-invokedynamic.html`. [Accessed: October 20, 2015], 2008.

29 Francisco Ortin, Patricia Conde, Daniel Fernandez-Lanvin, and Raul Izquierdo. The runtime performance of invokedynamic: An evaluation with a java library. *IEEE software*, (4):82–90, 2014.

30 OSGi Alliance. Semantic versioning – technical whitepaper. Technical report, OSGi Alliance, 2010.

31 Tom Preston-Werner. Semantic Versioning 2.0.0. `http://semver.org/`. [Accessed: October 20, 2015], 2010.

32 Steven Raemaekers, Arie van Deursen, and Joost Visser. Measuring software library stability through historical version analysis. In *Proceedings ICSM'12*. IEEE, 2012.

33 Steven Raemaekers, Arie Van Deursen, and Joost Visser. Semantic versioning versus breaking changes: a study of the maven repository. Technical report, Delft University of Technology, Software Engineering Research Group, 2014.

34 Ian Robertson. A hazard of covariant return types and bridge methods. `http://www.artima.com/weblogs/viewpost.jsp?thread=354443`. [Accessed: October 20, 2015], 2013.

35 John R Rose. Bytecodes meet combinators: invokedynamic on the jvm. In *Proceedings VMIL'09*. ACM, 2009.

36 Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

37 Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. An analysis of language-level support for self-adaptive software. *ACM Trans. Auton. Adapt. Syst.*, 8(2):7:1–7:29, July 2013.

38 Attila Szegedi. JEP 276: Dynamic Linking of Language-Defined Object Models. `http://openjdk.java.net/jeps/276`. [Accessed: November 10, 2015], 2015.

39 Ewan Tempero, James Noble, and Hayden Melton. How do java programs use inheritance? an empirical study of inheritance in java software. In *Proceedings ECOOP'08*. Springer, 2008.

40 The OSGi Alliance. OSGi core release 5. `http://www.osgi.org/Specifications`, 2012.

Appendices

A Dynamo DSL Grammar

```
kind     ::= '+' | '-'
role     ::= 'callsite' | 'target'
filters  ::= filter ( ',' filter )*
className ::= NAME
methodName ::= NAME
descriptor ::= '(' NAMES ')' NAME
filter   ::= kind role className ( '#' methodName descriptor? )?
NAMES    ::= ( NAME ( ',' NAME )* )+
```

`NAME` is used as defined in [22, sect 6], but with added support for the `*` and `?` wildcard characters.

B Dynamo Compiler Options

Table 9 Dynamo Compiler Options.

parameter	required	description
`-cp`	yes	Specify where to find user class files and annotation processors
`-d`	yes	Specify where to place generated class files
`-sourcepath`	yes	Specify where to find input source files
`-classic`	no	Classic compilation without enhancement - same as javac
`-encoding`	no	Specify character encoding used by source files
`-filter`	no	Specify which methods to enhance as defined in section 4.3
`-help`	no	Print instructions

Object Inheritance Without Classes

Timothy Jones[1], Michael Homer[2], James Noble[3], and Kim Bruce[4]

1 **Victoria University of Wellington, Wellington, New Zealand**
`tim@ecs.vuw.ac.nz`
2 **Victoria University of Wellington, Wellington, New Zealand**
`mwh@ecs.vuw.ac.nz`
3 **Victoria University of Wellington, Wellington, New Zealand**
`kjx@ecs.vuw.ac.nz`
4 **Pomona College, Claremont, California, USA**
`kim@cs.pomona.edu`

Abstract

Which comes first: the object or the class? Language designers enjoy the conceptual simplicity of object-based languages (such as Emerald or Self) while many programmers prefer the pragmatic utility of classical inheritance (as in Simula and Java). Programmers in object-based languages have a tendency to build libraries to support traditional inheritance, and language implementations are often contorted to the same end. In this paper, we revisit the relationship between classes and objects. We model various kinds of inheritance in the context of an object-oriented language whose objects are not defined by classes, and explain why class inheritance and initialisation cannot be easily modelled purely by delegation.

1998 ACM Subject Classification D.3.3 Classes and objects

Keywords and phrases Inheritance, Objects, Classes, Operational semantics

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.13

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.2.1.6`

1 Introduction

Class-based object-oriented languages have a simple story about the relationships between objects and the classes that create them: an object is an instance of a class [2]. A specialised, 'one-off' object is just an instance of a specialised, one-off, anonymous class [27]. Inheritance is between classes, and new objects are constructed and initialised by their classes.

This simple story comes at the expense of a more complicated story about classes — especially so if classes are themselves objects. Thirty years ago, Alan Borning identified eight separate roles that classes can play in most class-based object-oriented languages [4], each of these roles adding to the complexity of the whole language, which typically leads inexorably to various kinds of infinite regress in meta-object systems [21, 29, 44].

To address this problem, prototype-based object-oriented languages, beginning with Lieberman's work inspired by LOGO [32] and popularised by Self [51], adopted a conceptually simpler model in which objects were the primary concept, defined individually, without any classes. Inheritance-like sharing of state and behaviour was handled by delegation between objects, rather than inheritance between their defining classes. Special-purpose objects could be defined directly, while new objects could be created in programs by cloning existing objects.

```
method graphic(canvas) {
    object {
        method image { abstract }
        method draw { canvas.render(image) }
        var name
        canvas.register(self)
        draw // Local method request
    }
}
```

```
def amelia = object {
    inherit graphic(canvas)
    // Override using a field getter
    def image = images.amelia
    // Assign to inherited field
    name := "Amelia"
}
```

Figure 1 An example object constructor method and an inheriting object.

Emerald [3] went one step further and aimed to eschew all implicit sharing mechanisms, supporting neither inheritance nor delegation.

Programmers using object-based languages have found the need to reintroduce classes — several times over in many of these languages. The Emerald compiler, and later the Self IDE, added explicit support for class-style inheritance. Languages with object inheritance such as Lua [34], JavaScript [46], and Tcl [52] have a variety of libraries implementing classes in terms of objects. Most recently classes have been added explicitly to the recent ECMAScript standard [53], to bring some order to the profusion of libraries already offering classes.

The problem we address in this paper is precisely the tension between the conceptual simplicity of objects and the practical utility of classes: can a language be based conceptually on 'objects first' and include a relatively familiar notion of inheritance? In this paper we present models of eight inheritance mechanisms for a language without an inherent class construct: forwarding, delegation, concatenation, merged identity, uniform identity, multiple uniform identity, method transformations, and positional inheritance. The first three correspond to the foundational object-based models, while merged identity and uniform identity introduce more classical behaviour that parallels C++ and Java, and the remaining models introduce multiple object inheritance with different conflict resolution techniques.

We evaluate the tradeoffs between power and complexity of these models, particularly in their object initialisation semantics, and compare them to the behaviour of other languages, demonstrating that the typical class initialisation semantics are fundamentally at odds with prototypical object inheritance. We have also implemented all of the models in PLT Redex [17], making the models executable and allowing direct comparison of the differences in execution for any program.

2 Inheritance Without Classes

The term 'inheritance' typically refers to reuse relationships between classes, but there are also a number of 'objects-first' languages that eschew classes and permit reuse relationships directly between objects. Consider the example graphic method in Figure 1: the method constructs a fresh object with some methods and a mutable (var) field, and runs some initialisation code including a local call to the draw method before returning. If we can inherit from objects created by this method, what is the resulting program behaviour? Results vary for different interpretations of inheritance. We can inherit from the objects this method creates, or assign special semantics to methods which directly construct fresh objects like this and inherit from the method itself. Because of the presence of initialisation code in the class, these interpretations have different — and potentially unexpected — behaviours.

In its most basic form, inheritance permits reuse by providing a mechanism for an object to defer part of its implementation to an already implemented part of the program, but the reality is that there is much more to consider: the value of 'self' in method bodies and during initialisation, intended and accidental method overriding, whether objects are composed of several object identities in an inheritance chain or a single unified identity, the meaning of method requests which are not qualified with a receiver, and so on. We can draw different conclusions about the state of our program after amelia in Figure 1 is constructed, depending on which inheritance semantics is in play. We group these into our relevant concerns:

- **Registration**. Is the identity of a super-object stored during initialisation, either explicitly or through lexical capture, the same as the final object? This is clearly the intention of the call to register in graphic's initialisation.
- **Down-calls**. Can a method in a super-object call down into a method in a lower object? Can it do so during initialisation? The implementation of the draw method relies on a down-call to the image method.
- **Action at a Distance**. Can operations on an object implicitly affect another object? If the registered graphic object is different to amelia, what is the value of its name field after amelia is initialised?
- **Stability**. Is the implementation of methods in an object the same throughout its lifetime? Which image method will be invoked by the request to draw at the end of graphic? Can the structural type of an object change after it has been constructed?
- **Preëxistence**. Can an object inherit from any other object it has a reference to? Does amelia have to inherit a call to the graphic method, or will a preëxisting object suffice?
- **Multiplicity**. Can an object inherit from multiple other objects? If amelia also wished to have the behavior of another object, can a second inherit clause be added? If so, how are multiple methods with the same name resolved, and where are fields located?

We are also interested in the general semanticses of the inheritance systems, such as what order the parts of initialisation execute in, what visibility and security concerns arise, and how local method requests are resolved. We also point out particular curiosities, such as the absence of overloaded methods from super references and inheriting from definitions already inherited from some other parent, but these features are specific to individual models.

These are the concerns that we will judge in the following object inheritance models. While our graphic example clearly assumes some of these features in its implementation, these features are not universally desirable, and each provides certain abilities or guarantees at some cost. Our intention is to use these concerns to provide an accurate description of the tradeoffs involved with each inheritance model. Some of our models of inheritance attempt to interpret graphic as a class, but only in the sense that it is a factory that guarantees fresh graphic objects. We also compare the models to existing languages that use each form of inheritance, particularly JavaScript, which is capable of implementing all of the models.

3 Graceless

In order to provide a formal semantics for the various object inheritance systems, we first present a base model of the Grace programming language without inheritance, and then proceed to extend this base model in different ways to construct the different behaviours. Grace is a useful language to model object inheritance features in, as it is not class-based, and it does not permit mutation of an existing object's structure, so we can consider pure

Syntax

$$
\begin{array}{lcll}
e & ::= & o \mid r.m(\overline{e}) \mid m(\overline{e}) \mid \mathsf{self} \mid v \mid \boxed{\mid e; e \mid f \xrightarrow{x} \mid f \xleftarrow{x} e} & \textit{(Expression)} \\
S & ::= & \textbf{def}\ x = e \mid \textbf{var}\ x \mid \textbf{var}\ x := e \mid e & \textit{(Statement)} \\
M & ::= & \textbf{method}\ m(\overline{x})\ \{\, \overline{e;}\ e \,\} & \textit{(Method)} \\
m & ::= & x \mid x{:=} & \textit{(Method name)} \\
o & ::= & \textbf{object}\ \{\, \overline{M}\ \overline{S} \,\} & \textit{(Object expression)} \\
\sigma & ::= & \boxed{\cdot \mid \ell \mapsto \langle \overline{F}, \overline{M} \rangle, \sigma} & \textit{(Object store)}
\end{array}
\qquad
\begin{array}{lcll}
v & ::= & \textbf{done}\ \boxed{\mid \ell} & \textit{(Value)} \\
r & ::= & e & \textit{(Method receiver)} \\
f & ::= & \mathsf{self}\ \boxed{\mid \ell} & \textit{(Field receiver)} \\
F & ::= & \boxed{x \mapsto v} & \textit{(Field)} \\
s & ::= & \boxed{v/x \mid \mathsf{self}.m/m} & \textit{(Substitution)}
\end{array}
$$

Evaluation context

$$E ::= [] \mid E.m(\overline{e}) \mid v.m(\overline{v}, E, \overline{e}) \mid m(\overline{v}, E, \overline{e}) \mid E; e \mid \ell \xleftarrow{x} E$$

Figure 2 The grammar of the Graceless base model, with runtime-only components boxed.

object concerns without being overcome by 'open objects' with trivially mutable structure, as found in JavaScript.

Our core model resembles the existing Tinygrace language [28], but with more features, such that 'Tiny' is not an appropriate moniker. As the language is still not a complete implementation of Grace, we have opted to name it *Graceless*. We have modelled object references, mutable and immutable fields, and method requests unqualified by a receiver, in order to demonstrate the wide-ranging effects of changes to the behaviour of inheritance. Explicit types have been removed, in order to focus on the dynamic semantics.

The grammar for Graceless is provided in Figure 2. The boxed areas in the grammar represent forms which only exist at runtime, not in the user-side syntax of programs. As in Grace, we omit empty argument and parameter parentheses, making some method requests indistinguishable from local variables and field lookups per Meyer's uniform access principle [37]. Within an object are method definitions M and statements S. A statement is either a field declaration or an expression, the latter allowing method requests in object initialisation code. Field declarations are either constant **def**initions or **var**iable, and **var** fields may be declared without an initial value.

Expressions e in the user-side syntax are object expressions o, method requests either qualified or unqualified by a receiver, or the sentinel value **done**. At runtime, expressions can also include references ℓ to locations in the heap, sequences $e; e$, field fetch $\mathsf{self} \xrightarrow{x}$, and field assign $\mathsf{self} \xleftarrow{x} e$, with self eventually substituted for some ℓ. The field operations are distinguished from method calls $e.x$ and $e.x := e$.

Objects can be nested inside of each other (as method bodies) and unqualified requests can be made on the resulting local scope. We assume Barendregt's rule [1] for self references, such that each self variable introduced by nested object expressions is unique. Because method names appear in the local scope as well as the public interface of their directly surrounding object, we cannot assume Barendregt's rule that their names are unique — an object may need to shadow an outer object's method in order to conform to a given interface — so substitution cannot continue past a shadowing definition.

Graceless has two forms of substitution. The typical substitution $[v/x]e$ replaces the term x with the value v in the term e. A *qualifying substitution* $[\mathsf{self}.m/m]e$ replaces any local request

$\boxed{\langle \sigma, e\rangle \rightsquigarrow \langle \sigma, e\rangle}$

(E-Context)
$$\frac{\langle \sigma, e\rangle \rightsquigarrow \langle \sigma', e'\rangle}{\langle \sigma, E[e]\rangle \rightsquigarrow \langle \sigma', E[e']\rangle}$$

(E-Request)
$$\frac{\textbf{method}\ m(\overline{x})\ \{\, e\,\} \in \sigma(\ell)}{\langle \sigma, \ell.m(\overline{v})\rangle \rightsquigarrow \langle \sigma, [\ell/\mathsf{self}]\overline{[v/x]}e\rangle}$$

(E-Next)
$$\frac{}{\langle \sigma, v; e\rangle \rightsquigarrow \langle \sigma, e\rangle}$$

(E-Fetch)
$$\frac{}{\langle \sigma, \ell \xrightarrow{x}\rangle \rightsquigarrow \langle \sigma, \sigma(\ell)(x)\rangle}$$

(E-Assign)
$$\frac{}{\langle \sigma, \ell \xleftarrow{x} v\rangle \rightsquigarrow \langle \sigma(\ell)(x \mapsto v), \textbf{done}\rangle}$$

(E-Object)
$$\frac{\ell \text{ fresh} \qquad \overline{m} = \text{names}(\overline{M}, \overline{S}) \qquad \langle \overline{M_f}, \overline{e}\rangle = \text{body}(\overline{S})}{\langle \sigma, \textbf{object}\,\{\, \overline{M}\ \overline{S}\,\}\rangle \rightsquigarrow \langle \sigma(\ell \mapsto \langle \varnothing, \overline{[\mathsf{self}.m/m]}\overline{M}\ \overline{M_f}\rangle), [\ell/\mathsf{self}]\overline{[\mathsf{self}.m/m]}\overline{e}; \ell\rangle}\ \overline{m}\ \text{unique}$$

Auxiliary Definitions

$$\text{names}(\overline{\textbf{method}\ m(\overline{x})\ \{\, e\,\}}, \overline{S}) = \overline{m} \cup \overline{m_f} \qquad \textbf{where}\ \langle \overline{\textbf{method}\ m_f(\overline{y})\ \{\, e_f\,\}}, e\rangle = \text{body}(\overline{S})$$

$$\begin{aligned}
\text{body}(\varnothing) &= \langle \varnothing, \varnothing\rangle \\
\text{body}(\textbf{def}\ x = e, \overline{S}) &= \langle \text{accessors}(\textbf{def}, x, y)\ \overline{M}, \mathsf{self} \xleftarrow{y} e\ \overline{e}\rangle && \textbf{where}\ y \text{ fresh and } \langle \overline{M}, \overline{e}\rangle' = \text{body}(\overline{S}) \\
\text{body}(\textbf{var}\ x, \overline{S}) &= \langle \text{accessors}(\textbf{var}, x, y)\ \overline{M}, \overline{e}\rangle && \textbf{where}\ y \text{ fresh and } \langle \overline{M}, \overline{e}\rangle' = \text{body}(\overline{S}) \\
\text{body}(\textbf{var}\ x := e, \overline{S}) &= \langle \text{accessors}(\textbf{var}, x, y)\ \overline{M}, \mathsf{self} \xleftarrow{y} e\ \overline{e}\rangle && \textbf{where}\ y \text{ fresh and } \langle \overline{M}, \overline{e}\rangle' = \text{body}(\overline{S}) \\
\text{body}(e, \overline{S}) &= \langle \overline{M}, e\ \overline{e}\rangle && \textbf{where}\ \langle \overline{M}, \overline{e}\rangle' = \text{body}(\overline{S})
\end{aligned}$$

$$\text{accessors}(\textbf{def}, x, y) = \textbf{method}\ x\ \{\, \mathsf{self} \xrightarrow{y}\,\}$$
$$\text{accessors}(\textbf{var}, x, y) = \textbf{method}\ x\ \{\, \mathsf{self} \xrightarrow{y}\,\}\ \textbf{method}\ x{:=}(z)\ \{\, \mathsf{self} \xleftarrow{y} z\,\}$$

Figure 3 Term reduction.

$m(\overline{e})$ with the qualified form $\mathsf{self}.m(\overline{e})$ in the term e. Both forms of substitution are ended by a shadowing definition, which can be either an adjacent method name or a surrounding parameter definition: substitutions into an object expression $[v/x]o$ and $[\mathsf{self}.m/m]o$ do not modify o if it contains a method x or m respectively, and the substitutions into a method $[v/x]M$ and $[\mathsf{self}.x/x]M$ do not modify the method body if M has a parameter x. Because the local variable x and the local method request x are indistinguishable, the ordering of the substitution matters: $[\mathsf{self}.x/x][v/x]x$ produces v, but $[v/x][\mathsf{self}.x/x]x$ produces $\mathsf{self}.x$.

The reduction judgment $\langle \sigma, e\rangle \rightsquigarrow \langle \sigma', e'\rangle$ is defined in Figure 3, indicating an expression e with store σ is reduced to an expression e' with a potentially modified store σ'. Rule E-Context uses the evaluation context to perform congruence reduction, and Rule E-Next evaluates to the next expression in a sequence when the current one has finished evaluating. Rules E-Fetch and E-Assign handle operations on a field store. The store access $\sigma(\ell)(x)$ looks up the field x in the object at location ℓ, getting stuck if the field has not been initialised yet. The store update $\sigma(\ell)(x \mapsto v)$ sets the field x to the value v in the object at location ℓ, introducing a new field if one was not already present.

Rule E-Request processes requests by looking up the corresponding method in the receiver. In the method body, it substitutes both the arguments for parameter names, and the receiver for the name self. Rule E-Object takes an object expression and builds a corresponding object in the store, with no fields, the methods in the object expression, and

Extended Syntax

$I ::= \textbf{inherit}\ e\ \boxed{\overline{s}}$ *(Inherit clause)*

$o ::= \cdots \mid \textbf{object}\,\{\, I\ \overline{M}\ \overline{S}\,\}$ *(Object expression)*

$s ::= \cdots \boxed{\mid (\ell\ \textbf{as}\ \mathsf{self})/\mathsf{super}}$ *(Substitution)*

$v ::= \cdots \boxed{\mid (\ell\ \textbf{as}\ v)}$ *(Value)*

$r ::= \cdots \mid \mathsf{super} \boxed{\mid (\ell\ \textbf{as}\ e)}$ *(Receiver)*

$\boxed{\langle \sigma, e\rangle \rightsquigarrow \langle \sigma, e\rangle}$

(E-Request/Super)

$$\frac{\textbf{method}\ m(\overline{x})\ \{\, e\,\} \in \sigma(\ell_\uparrow)}{\langle \sigma, (\ell_\uparrow\ \textbf{as}\ \ell_\downarrow).m(\overline{v})\rangle \rightsquigarrow \langle \sigma, [\ell_\downarrow/\mathsf{self}]\overline{[v/x]}e\rangle}$$

(E-Inherit/Context)

$$\frac{\langle \sigma, e\rangle \rightsquigarrow \langle \sigma', e'\rangle}{\langle \sigma, \textbf{object}\,\{\,\textbf{inherit}\ e\ \overline{s}\ \overline{M}\ \overline{S}\,\}\rangle \rightsquigarrow \langle \sigma', \textbf{object}\,\{\,\textbf{inherit}\ e'\ \overline{s}\ \overline{M}\ \overline{S}\,\}\rangle}$$

(E-Inherit)

$$\frac{\langle \overline{F}, \overline{M_\uparrow}\rangle = \sigma(\ell) \qquad \overline{M'_\uparrow} = \mathrm{override}(\overline{M_\uparrow}, \mathrm{names}(\overline{M}, \overline{S}))}{\langle \sigma, \textbf{object}\,\{\,\textbf{inherit}\ \ell\ \overline{s}\ \overline{M}\ \overline{S}\,\}\rangle \rightsquigarrow \langle \sigma, \textbf{object}\,\{\,\overline{M'_\uparrow}\ \overline{[s]}[(\ell\ \textbf{as}\ \mathsf{self})/\mathsf{super}](\overline{M}\ \overline{S})\ \rangle\,\}}$$

Auxiliary Definitions

$$\mathrm{override}(\varnothing, \overline{m}) = \varnothing$$

$$\mathrm{override}(\textbf{method}\ m(\overline{x})\ \{\, e\,\}\ \overline{M}, \overline{m}) = \begin{cases} m \in \overline{m} & \mathrm{override}(\overline{M}, \overline{m}) \\ m \notin \overline{m} & \textbf{method}\ m(\overline{x})\ \{\, e\,\}\ \mathrm{override}(\overline{M}, \overline{m}) \end{cases}$$

Figure 4 Object inheritance extension.

the generated getter and setter methods for the fields. The rule also converts the fields in the object expression into a series of assignments, which ultimately result in the new reference. The internal field names are (globally) fresh to avoid having to worry about overridden names under inheritance. Both the methods and the assignments have the relevant qualifying substitutions applied, and the body of the object has self bound to the new reference.

Note that we could bind self in the bodies of an object's methods either in the Rule E-Object (early binding, when the object is allocated) or in Rule E-Request (late binding, when a method is requested). In this model, either choice produces the same behaviour. As we add inheritance to Graceless, our choice to use late binding in the base model will matter: it is trivial to overwrite by also using early binding, but not the other way around.

4 Object Inheritance

We now extend Graceless with various implementations of object inheritance, and consider the complexity and impacts of the changes. The extensions are presented in a rough ordering of implementation complexity. These represent the three foundational strands of object inheritance: forwarding, as used in E; delegation, as found in JavaScript [53], Lua [25], and Self [51, 10]; and concatenation, as in Kevo [48, 49] and numerous libraries and idioms for languages with open objects.

The extended syntax for object inheritance is given in Figure 4. The extension introduces two new components of user-facing syntax: the bodies of object expressions may now begin with an **inherit** e clause, and requests can now be qualified by the special variable super.

These two components each result in a runtime complication. This intermediate extension of Graceless already implements a form of object inheritance similar to concatenation with all fields reset to uninitialised, but we do not know of any language with this behaviour, nor can we see why it would be desirable.

Inheritance introduces the methods from the super-object into the local scope of the inheriting object using an 'up, then out' rule: inherited definitions take precedence over those introduced in a surrounding scope. Because the inherit clause contains an arbitrary expression, we might not know what the names are that the clause will introduce. To counter this, substitutions are delayed by an inherit clause in an object expression: while the substitution will transform the expression in the clause itself, it *will not* proceed into the body of the object expression, and gets 'stuck' on the clause instead. Once the expression in an inherit clause is resolved to an object reference, the substitution can proceed into the body of the surrounding object expression, where it may be removed by shadowing.

Although there is an explicit super-object in this model of inheritance, making a request to super is not the same as making a direct request to the super-object, as the value of self will be bound to the inheriting object. At runtime, the variable super is substituted for a special 'up-call' receiver (ℓ **as** self), which indicates the method to call should be sourced from the object at location ℓ, but self in the body of that method should be bound to the eventual value of self at the site of the request.

The evaluation context has not been extended as expected, to avoid issues in the future with constraints on what can be evaluated in an inherit clause. Evaluation in an inherit clause is instead handled by Rule E-Inherit/Context, which is modified by future models. Rule E-Inherit transforms an object expression with an inherit clause into one without, by copying the methods from the super-object which are not overridden by a method with the same name into the body of the inheriting object (directly copying the methods has the same behaviour as creating new methods which delegate to the super-object with the same arguments, or searching through a chain of objects for the method, which is representative of typical object inheritance implementations). It also applies the delayed substitutions to the body, after substituting super for an up-call receiver to the inherited object. Rule E-Request/Super simply applies an up-call as described.

By making subtle modifications to the existing rules in this extended form of Graceless, we can produce models for various implementations of object inheritance.

4.1 Forwarding

Under forwarding, inherited methods are simply redirected to the super-object. The super-method receives the same arguments and **self** binding. In the example from earlier, if amelia receives a request for the draw method, which is not implemented directly inside of amelia, the request is passed on to the graphic super-object instead. The value of **self** in the resulting invocation is the identity of the super-object: in this example, the draw method crashes, complaining the graphic has not implemented its image method, because the local request to image has been resolved to the graphic object and not passed back down to amelia.

The modification to the existing Graceless dynamic semantics to implement forwarding is provided in Figure 5. The modification makes one subtle change (highlighted) to the Rule E-Object, by early-binding the value of self when an object is created in both its methods and field accessors. The result of this change is that the late-binding of self in requests (both normal and to super) no longer achieves anything, because self has already been bound to the object that the method or field originally appeared in. Any forwarded request behaves as though the original object had received the request directly.

$$\text{(E-Object/Forwarding)}\quad \frac{\ell \text{ fresh} \qquad \overline{m} = \text{names}(\overline{M}, \overline{S}) \qquad \langle \overline{M_f}, \overline{e} \rangle = \text{body}(\overline{S})}{\langle \sigma, \textbf{object}\ \{\ \overline{M}\ \ \overline{S}\ \}\rangle \rightsquigarrow \langle \sigma(\ell \mapsto \langle \varnothing, [\ell/\mathsf{self}](\overline{[\mathsf{self}.m/m]M}\ \ \overline{M_f})\rangle), [\ell/\mathsf{self}]\overline{[\mathsf{self}.m/m]e};\ \ell\rangle}\ \overline{m}\text{ unique}$$

Figure 5 Forwarding modifications.

The value of **self** is always the object a method was defined in, so down-calls are not possible; similarly, the value of **self** during initialisation is the distinct identity of the super-object, so registration cannot occur then. In this model, an object can only inherit from another before it has run any of its own initialisation, so every object has a stable structure, but one object may be inherited many times. While this model does not permit multiple super-objects, it would only require updating Rule E-Inherit to include methods from multiple inherited objects, with some arbitrary mechanism to resolve multiply-defined methods (such as placing significance on the order of the inherit clauses, or requiring that a multiply-defined method be overridden in the inheriting object).

There is no concept of confidential access of methods between implementations, as a forwarded request is sent as a regular request, and so will only be handled by the public interface of the super-object. Forwarding also does not permit downcalls: an inherited object cannot invoke an inheriting object's method. On the other hand, an object can forward messages to a preëxisting object, and many forwarding objects can share a single target. Inherited fields are shared between all inheriting objects, and the mutation of an inherited field will implicitly affect the super-object and all of its heirs.

In the E language, there is no explicit self reference, as all object definitions are explicitly named. The authors of E refer to the language's inheritance mechanism as "delegation", but in the absence of self references the behaviour aligns with what we have called forwarding.

```
def graphic { to draw() { canvas.render(graphic.image()) } }
def amelia extends graphic { to image() { images.cat() } }
```

Even though amelia defines a method image, the call in draw clearly looks for the method in the graphic object. Methods cannot be requested on the local scope, so the receiver must always be explicit. In order to achieve down-calls, the inheriting object must explicitly be passed to the super-object, which is a standard pattern for simulating class behaviour in E.

4.2 Delegation

Delegation is an implementation of object inheritance that aimed to be at least as powerful as inheritance, if not more so [32, 51, 33]. Delegation has a subtle distinction from forwarding: a **self** request in a method called under delegation goes *back to the original object*, while under forwarding a **self** request to a delegatee will be handled only by that delegatee. This allows delegation to support down-calls, where forwarding cannot.

In the previous example, it was not safe to request the draw method on amelia under forwarding, as the implementation of draw expects to be able to see an overridden implementation of the image method. Under delegation, this now works as expected: draw executes with **self** as amelia, so the local request to image calls down into amelia and successfully retrieves the image of Amelia.

The modification to the existing Graceless dynamic semantics to implement delegation is provided in Figure 6. Like forwarding, the modification early-binds the value of self in the

(E-Object/Delegation)

$$\frac{\ell \text{ fresh} \qquad \overline{m} = \text{names}(\overline{M}, \overline{S}) \qquad \langle \overline{M_f}, \overline{e} \rangle = \text{body}(\overline{S})}{\langle \sigma, \textbf{object}\ \{\ \overline{M}\ \ \overline{S}\ \}\rangle \rightsquigarrow \langle \sigma(\ell \mapsto \langle \varnothing, \overline{[\mathsf{self}.m/m]M}\ \ [\ell/\mathsf{self}]\ \overline{M_f}\rangle), [\ell/\mathsf{self}]\overline{[\mathsf{self}.m/m]e};\ \ell\rangle}\ \overline{m}\text{ unique}$$

Figure 6 Delegation modifications.

Rule E-Object, but this time *only in the field accessors*. The value of self remains late-bound in the bodies of methods to the receiver of a method request, allowing super-objects to perform down-calls to methods implemented in a sub-object. Fields are shared between the original object and any inheriting objects, and, as under forwarding, mutation of any field which originated in a super-object is reflected in all of its heirs.

Delegation makes no requirement of freshness, which combined with down-calls produces a further complication to information hiding in a language with confidential methods that can only be called on **self**. These methods may provide access to secret data or capabilities, and the ability to access them from arbitrary code could be a security concern. Delegation to preëxisting objects opens the way for the 'vampire' problem: any object to which a reference has been obtained can be taken over and fully controlled from the outside, merely by defining a new child object. If access to confidential methods is instead *not* provided to delegators, simulating common classical patterns becomes difficult or impossible, and an odd asymmetry is introduced: the delegator can override an inherited method, changing its behaviour, but has no access to use the overridden method in its own implementation.

Unlike forwarding, delegation permits down-calls after the object has been constructed, but *not during initialisation*. Objects under delegation cannot perform registration during initialisation, as a captured **self** reference in a super-object refers to that super-object, which may have no other references and will not have access to any overriding definitions from the child. These two limitations of object initialisation are the major barriers to simulating the typical behaviour of class-based inheritance under delegation.

Object structure and behaviour is not stable during construction, as new methods may appear on **self** and existing methods may have different implementations depending on the stage of inheritance. Like forwarding, delegation permits inheritance from a preëxisting object; if this is allowed, stability does not exist after construction either. Delegation, like forwarding, can be easily extended to multiple inheritance by introducing multiple inherit clauses and some resolution of multiply-defined methods.

Including the inherit clause in the object constructor ensures that objects can delegate either to a preëxisting prototype object or construct a new object with custom arguments and delegate to that, as an equivalent to calling a super-constructor in a class system. This distinguishes it from the prototypical inheritance in JavaScript, which requires that the prototype property of the constructor be set before any inheriting object is constructed. In Self, object expressions can set their parents as a fresh object constructor call:

```
( | parent* = factory new. | )
```

This has the same semantics in terms of **self** binding in the super-constructor as presented in our model of delegation.

$$\frac{\langle \overline{x \mapsto v}, \overline{M_\uparrow} \rangle = \sigma(\ell) \qquad \overline{M'_\uparrow} = \mathrm{override}(\overline{M_\uparrow}, \mathrm{names}(\overline{M}, \overline{S}))}{\langle \sigma, \textbf{object}\ \{\ \textbf{inherit}\ \ell\ \overline{s}\ \overline{M}\ \overline{S}\ \}\rangle \rightsquigarrow \langle \sigma, \textbf{object}\ \{\ \overline{M'_\uparrow}\ \overline{[s]}[(\ell\ \textbf{as}\ \mathsf{self})/\mathsf{super}](\overline{M}\ \overline{\mathsf{self} \xleftarrow{x} v}\ \overline{S})\ \rangle\ \}}\ (\text{E-Inherit/Concatenation})$$

Figure 7 Concatenation modification.

4.3 Concatenation

Concatenation is an alternative approach to both forwarding and delegation that aimed to have the power of inheritance without the drawbacks of delegation [48, 49]. Under concatenation, one object inherit from another by (conceptually) taking a shallow copy of the methods and fields of its parent into itself, and then appending local overriding definitions. Concatenation supports down-calls, but unlike delegation does not allow subsequent changes in either the parent or the child to affect each other.

The modification to the existing Graceless dynamic semantics to implement concatenation is provided in Figure 7. Unlike the previous two models, this modification instead changes the Rule E-Inherit, to copy the fields from the super-object into the inheriting object as assignments. The existing late-binding of self in methods is sufficient to provide the desired behaviour: any inherited method executes in the context of the inheriting object, and any request to an inherited field accessor will access the copied field in the inheriting object as well. The only relationship the inheriting object has with its super-object is explicit up-calls, but it is impossible to access or modify the state of the super-object without explicitly referring to it through an existing reference. The resulting behaviour is equivalent to delegating to a clone of the super-object.

Concatenation also makes no requirement of freshness. Concatenation with preëxisting objects does not quite permit the 'vampirism' of delegation, but does allow 'mind reading': any confidential state in an object can be read simply by inheriting from it, but the existing object cannot be manipulated by the child. Unlike the two previous models, mutations to inherited fields do not cause action at a distance, as the mutation will always affect a field in the actual receiver of the request (even for super-calls). With lexical scoping, the two objects are also not as independent as they seem: methods exist in the same scope in both objects, and any lexically captured state is shared between the two.

Like delegation, concatenation permits down-calls after the object has been constructed, but still not during construction. Concatenation does not allow registration, as a captured **self** reference in a super-object refers to the parent. Object structure and behaviour is not stable during construction; as for delegation, if inheritance from preëxisting objects is allowed then stability does not exist afterwards either. Concatenation can be straightforwardly extended to multiple inheritance by inserting the contents of each parent into the child, with some resolution of multiply-defined methods.

Objects with mutable structure can trivially implement concatenation, by manually copying the structure of the inherited object into the inheriting one. JavaScript complicates this story with unenumerable properties, implicit field accessors, and existing delegation relationships, but for the most part this is a valid implementation of concatenation:

```
for (var name in inherited) { inheriting[name] = inherited[name]; }
```

It is also possible to use mutable object structure to implement either forwarding or delegation, by assigning methods (or field accessors) to the inheriting object that directly forward or delegate to the inherited object, as in our models. In a JavaScript constructor:

$$\frac{\langle \sigma, e\rangle \rightsquigarrow \langle \sigma', e'\rangle \qquad e = v.m(\overline{v}) \implies e' = \overline{e}; o}{\langle \sigma, \textbf{object}\,\{\,\textbf{inherit}\ e\ \overline{s}\ \overline{M}\ \overline{S}\,\}\rangle \rightsquigarrow \langle \sigma', \textbf{object}\,\{\,\textbf{inherit}\ e'\ \overline{s}\ \overline{M}\ \overline{S}\,\}\rangle}\ (\text{E-Inherit/Context})$$

Figure 8 Fresh inheritance modification.

```
var self = this;
this.foo = function () { self.bar(); };
```

If the foo method is called on a sub-object, the call to bar is guaranteed to not perform a down-call, because self is bound directly to the original object.

JavaScript's built-in object inheritance otherwise works the same way as Graceless delegation, but field assignments are directly available in the language instead of only through accessor methods. The result is that an object shares each field of its inherited object, but when a field is assigned to it, the field is unique to that object, shadowing the inherited one. In any language where the objects have mutable structure it is necessary to retain a parent reference in order to implement delegation, in order to accurately defer to the current implementation of a parent object. Implementing more complicated forms of inheritance in JavaScript, such as with traits or mixins, tends to involve combining the built-in delegation alongside manual concatenation.

5 Emulating Classes

The inheritance models in the previous section represent the three foundational strands of purely object-based inheritance. Class-based languages tend to provide different semantics, and programmers and language designers may wish to use those behaviours, or may unconsciously expect object-based languages to behave similarly.

It is possible to construct object-based models that parallel many of these classical behaviours. In some languages with very flexible semantics, such as JavaScript and Lua, libraries exist to provide "classes" as a second-class construct by mutating objects or leveraging specially-constructed objects with the existing inheritance systems [46, 34]. The two models in this section approximate the inheritance behaviour of C++ (Section 5.1) and Java (Section 5.2) in an object-based system. They remain purely object-based, but trade off some of the flexibilities of the object inheritance models for the classical functionalities they provide, such as registration, down-calls, and stability.

The new models use as their base a further extension to Graceless inheritance, provided in Figure 8, which enforces that the inherited object is a 'fresh' reference. This extension modifies Rule E-Inherit/Context so that method requests ready to be computed directly in an inherit clause may only be reduced if they syntactically return an object expression. This modification to the rule still permits the object constructor to then be resolved to a reference before handling the inherit clause.

The requirement of strict syntactic freshness is not required for merged identity, but it prevents potentially dangerous manipulation of object identity from outside of an object's creator. Merged identity builds directly on the new extension, while uniform identity further modifies the context rule to also prevent an object expression from being computed in an inherit clause.

$$\text{(E-Inherit/Merged)}$$
$$\frac{\langle \overline{F}, \overline{M_\uparrow} \rangle = \sigma(\ell) \qquad \ell_\uparrow \text{ fresh} \qquad \overline{m_\downarrow} = \text{names}(\overline{M}, \overline{S}) \qquad \overline{M'_\uparrow} = \text{override}(\overline{M_\uparrow}, \overline{m_\downarrow}) \qquad \overline{m} = \overline{m_\downarrow} \cup \text{names}(\overline{M'_\uparrow}, \varnothing) \qquad \langle \overline{M_f}, \overline{e} \rangle = \text{body}(\overline{S}) \quad \overline{m} \text{ unique}}{\begin{array}{c} \langle \sigma, \textsf{object}\ \{\ \textsf{inherit}\ \ell\ \overline{s}\ \overline{M}\ \overline{S}\ \} \rangle \rightsquigarrow \\ \langle \sigma(\ell_\uparrow \mapsto \langle \varnothing, \overline{M_\uparrow} \rangle)(\ell \mapsto \langle \overline{F}, \overline{M'_\uparrow}\ \overline{[s][\textsf{self}.m/m]}[(\ell_\uparrow\ \textsf{as self})/\textsf{super}]\overline{M}\ \overline{M_f} \rangle), \\ \overline{[s]}[\ell/\textsf{self}]\overline{[\textsf{self}.m/m]}[(\ell_\uparrow\ \textsf{as self})/\textsf{super}]\overline{e};\ \ell \rangle \end{array}}$$

Figure 9 Merged identity modification.

5.1 Merged Identity

In this model, an inheriting object takes over the identity of its parent, 'becoming' that object but putting in place all of its own method definitions, in effect the reverse of concatenation. The parent object is constructed and initialised before the child, and then mutated at the point of inheritance. Rather than mutating preëxisting objects, inheritance must now occur from a fresh object, one newly created and immediately returned from a method call. Without the requirement of freshness, objects can directly steal the identity of any existing object: the resulting 'body-snatchers' problem is substantially worse than delegation's vampirism, which can only control objects internally. Overridden methods from the parent remain accessible through **super**, and will always execute with the final identity of the object.

Merged identity provides essentially the C++ inheritance behaviour: the apparent type of the object changes during construction, as each layer of inheritance is processed. After object construction is complete, down-calls resolve to their final overridden method, but until then they obtain the most recent definition from ((great-)grand)parent to child.

The use of **self** for registration in graphic will now correctly store the value of amelia when initialising the super-object. Although the object is *not* amelia when it is registered, amelia *becomes* the object that was registered once it has finished initialising, merging the two identities together. The initialisation of amelia will still fail at the local request of draw, because amelia's overriding of the abstract method has not yet been merged into the identity of the object.

The modification to Graceless with fresh inheritance to implement merged identity is provided in Figure 9. The Rule E-Inherit/Merged overwrites the existing Rule E-Inherit. Where the old rule resulted in a new object expression with some of the methods from the super-object, this new rule skips straight to returning the resulting reference. This is necessary because the resulting reference is not fresh: it is the reference of the inherited object. The inherited object is updated with the new methods in the store, including keeping all of the old field values, but removing overridden methods.

The new rule is a combination of the behaviour of the old E-Inherit and Rule E-Object. We have highlighted the changes to their combined behaviour. There is no longer an object which corresponds to super to perform an up-call substitution on the body of the object as in Rule E-Inherit, so rather than creating a fresh location for the resulting object, we create a fresh location for the super-object instead. The super-object has all of the methods of the inherited object before it was modified, while the merged object retains its own fields instead of beginning with an empty field store. All of the models which emulate classes with super-references create these 'part-objects', which exist purely to store the super-methods for super calls, never have any fields of their own, and are only ever referenced under an alias for the 'real' object that is bottom-most in the hierarchy.

Like delegation, merged identity enables down-calls after, but not during, initialisation. Unlike delegation, a single identity is preserved throughout the construction process, so registration is possible. It does not provide classical stability during initialisation, but once objects are complete they cannot change again. The meta-mutation of merging objects means that it unsafe to permit inheritance from arbitrary objects, so only freshly-constructed objects are valid parents. Merged identity is really the only model presented in this paper that does not lend itself well to multiple inheritance, because only a single parent can have its identity preserved.

As a reverse form of concatenation, merged identity is just as easy to implement for objects with mutable structure. A constructor can call into the super-constructor, then concatenate its own definitions into the resulting object. Copying the original object beforehand ensures that a super-reference is still available.

5.2 Uniform Identity

The uniform identity design is a direct attempt to match as closely as possible the behaviour of a typical class-based inheritance system, but based on objects rather than on classes. In this design, the *first* observable action is to create a new object identity in the bottom-most 'child' object constructor; this exactly mirrors the merged identity design, which creates a single identity of the *topmost* parent. All inherited declarations are assembled in a single object associated with this identity, but no fields are initialised and no inline code run until the direct inheritance completes. Inheritance occurs with the identity "passed along": declarations are attached to the original object, without initialisation, until the topmost object with no parent is reached. Finally, the initialisation code runs from top to bottom.

All initialisation occurs in the context of the final object, including its behaviour. No new methods or overrides are added visibly during construction. While initialisation code always sees objects as a consistent type, it may observe uninitialised fields, including constant fields. We model uninitialised field access as a fatal error, and leave static detection to a higher layer in the language. This model essentially aligns with the semantics of Java-like languages.

The behaviour of amelia is the same as in the merged identity semantics, except that the local request to draw in the initialisation of graphic now successfully down-calls into amelia's implementation, as the object *is* amelia during all initialisation in the hierarchy. The result is still an error, as amelia's initialisation of the image field has not yet been executed. Super-constructors may still encounter uninitialised fields that will be assigned to in some sub-constructor, particularly when methods can be overridden by fields.

The modification to Graceless with fresh inheritance to implement uniform identity is provided in Figure 10. As with merged identity, the Rule E-Inherit/Uniform overwrites the existing Rule E-Inherit. The Rule E-Inherit/Context has been further refined to also prevent the evaluation of object expressions directly inside an inherit clause.

Where Rule E-Inherit/Merged was a modified application of E-Inherit and then E-Object, the uniform identity modification applies this in reverse. As the body of the inherit clause is an object expression, we have to apply many of the same processes for reducing a regular object expression, but then move the sequence of expressions resulting from the body of the super-object down into the body of the inheriting object. The evaluation *does* create a new object reference for the super-object, but only for up-calls to super. All fields will end up in the ultimate inheriting object.

Because the inherited object expression has not been processed into a runtime object, the rule needs to manually build its field methods and body. The substitutions to local definitions $\overline{m}$ need to be applied to the inherited methods (both those in the store, and the

$$\frac{\langle \sigma, e\rangle \rightsquigarrow \langle \sigma', e'\rangle \qquad e = v.m(\overline{v}) \implies e' = \overline{e}; o \qquad e \neq o}{\langle \sigma, \textsf{object}\,\{\,\textsf{inherit}\ e\ \overline{s}\ \overline{M}\ \overline{S}\,\}\rangle \rightsquigarrow \langle \sigma', \textsf{object}\,\{\,\textsf{inherit}\ e'\ \overline{s}\ \overline{M}\ \overline{S}\,\}\rangle}\ \text{(E-Inherit/Context)}$$

$$\frac{\begin{array}{c}\ell \text{ fresh} \qquad \overline{m} = \text{names}(\overline{M}, \overline{S}) \qquad \overline{M_\uparrow} = [\textsf{self}.m/m]\overline{M} \\ \langle \overline{M_f}, \overline{e}\rangle = \text{body}(\overline{S}) \qquad \overline{m_\downarrow} = \text{names}(\overline{M_\downarrow}, \overline{S_\downarrow}) \qquad \overline{M'_\uparrow} = \text{override}(\overline{M_\uparrow}\ \overline{M_f}, \overline{m_\downarrow})\end{array}}{\begin{array}{c}\langle \sigma, \textsf{object}\,\{\,\textsf{inherit}\ \textsf{object}\,\{\,\overline{M}\ \overline{S}\,\}\ \overline{s}\ \overline{M_\downarrow}\ \overline{S_\downarrow}\,\}\rangle \rightsquigarrow \langle \sigma(\ell \mapsto \langle \varnothing, \overline{M_\uparrow}\ \overline{M_f}\rangle),\\ \textsf{object}\,\{\,\overline{M'_\uparrow}\ \overline{[s]}[(\ell\ \textsf{as self})/\textsf{super}]\overline{M_\downarrow}\ \ \overline{[\textsf{self}.m/m]e}\ \ \overline{[s]}[(\ell\ \textsf{as self})/\textsf{super}]\overline{S_\downarrow}\,\}\rangle\end{array}}\ \overline{m}\ \text{unique}$$

(E-Inherit/Uniform)

Figure 10 Uniform identity inheritance modification.

non-overridden methods in the inheriting object). The same substitution into the body of the inheriting object occurs as before, but now self is not bound in the body: self will be bound by the ultimate application of Rule E-Object. The substitutions applied simultaneously to $M_\downarrow$ and $S_\downarrow$ in the inheriting object by Rule E-Inherit now have to be split between the two, as the super-body e appears in between, but it amounts to the same behaviour.

Uniform identity permits down-calls both after and during construction, as well as registration, as the identity and structure of the object are both constant, also guaranteeing stability. It does not allow inheriting from preëxisting objects, instead requiring that parents be fresh. It does not support multiple inheritance, but there is a logical extension to do so.

Constructors have a special role in JavaScript, and must have their prototype property set before they are used to construct a new object. As a result, inheriting from a preëxisting prototype object is simple, but inheriting from another constructor, particularly one that requires arguments specific to a particular inheriting object, is more difficult. Object.create allows the creation of an object from a prototype without actually invoking the constructor:

```
function Bar(arg) { Foo.call(this, arg); }
Bar.prototype = Object.create(Foo.prototype);
```

The ability to bind the value of **this** in a function call allows JavaScript to implement uniform identity, ensuring that the initialisation code in a super-constructor can be executed in the context of the inheriting object instead of creating a fresh object and inheriting from that. JavaScript's class syntax, introduced by ECMAScript 2015, is just sugar for this behaviour.

The most natural forms of inheritance in JavaScript are the built-in single delegation and, by extension, single uniform inheritance, now codified directly in the language with the class syntax. Our primary concern is that without the particular dynamism of the necessary JavaScript features, simulating class initialisation is impossible without reinterpreting factory methods as constructors with special semantics, at which point the language has arguably just implemented classes.

6 Multiple Inheritance

Reusing behaviour from multiple parents is a widespread desire, but is less commonly supported in language designs in practice. In this section we show three logical extensions enabling multiple object inheritance. The first extends the uniform identity model with multiple **inherit** statements. The second is a separate model following the tradition of trait systems, where method names are unique in any object. The third extends any of the

base models with the ability to include multiple **inherit** statements in an object, processed imperatively. All of these extensions process inherit clauses as under uniform identity, but they could all be simplified to perform inheritance from preëxisting objects as per forwarding, delegation, or concatenation. We have demonstrated this by constructing all possible combinations in our Redex implementation.

All systems enable code reuse from arbitrarily-many parents. All parents are treated symmetrically in the first two models, but further restrictions or privileges could be accorded to some parents without substantially affecting the models. In order to handle conflicts in symmetric inheritance, methods can be **abstract** in their body, which causes an error if that method is called. Alternatively, the implementations could ban the construction of an object with abstract methods, but this would only be valid for uniform identity, as the overriding of a concrete implementation would not take effect until the object was inherited under any of the other interpretations.

6.1 Multiple Uniform

The multiple uniform model allows a sequence of **inherit** statements to appear at the start of an object body. Each statement includes an **as** name clause, which binds name locally to have **super** semantics with regard to that inheritance tree. The single-inheritance uniform identity model is recovered with **inherit** parent **as super**. When the same method name is inherited from multiple parents, none has priority and an abstract method of that name is inserted instead. The programmer must provide a local override calling the version from a particular named parent if desired. All methods are collected and installed before any initialisation code from the object bodies executes, so a consistent set of method implementations is seen throughout the initialisation.

The implementation of multiple uniform identity is presented in Figure 11, as an extension to Graceless inheritance with the caveat that super is no longer a special form of receiver. The Rule E-Inherit/Context implements fresh inheritance in objects with potentially multiple inherit clauses, ensuring each clause is evaluated in order.

Rule E-Inherit/Multiple is essentially the same as Rule E-Inherit/Uniform, but processing multiple inherit clauses at once (hence the extra multiplicities for many of the bindings). Once all of the super-methods are collected, conflicting methods are resolved with the join auxiliary function that, for each unique method name in the collection of methods, accepts exactly one concrete implementation of a method with that name and removes all of the abstract implementations, or provides a single abstract method with that name and removes all other implementations.

Multiple uniform supports registration and downcalls exactly as in uniform identity. It supports multiple inheritance from freshly-created parents, and is stable because methods are collected first. These are all properties of uniform inheritance: applying this modification to the simpler object inheritance models each retains their own particular properties as well.

6.2 Method Transformations

Multiple inheritance under method transformations resembles trait-like composition of objects, representing object values as a single mapping of method names to methods, with no equivalent to **super** in the previous designs. Instead, an **inherit** statement can have any number of **alias** or **exclude** clauses associated, which respectively create an alternative name for an inherited method and exclude its implementation. If a method is overridden locally and still needs to be accessed, the inherited method can be aliased to a different name and

Extended Syntax

$I ::= \textbf{inherit}\ e\ \textbf{as}\ x$ *(Inherit clause)* $\quad o ::= \textbf{object}\ \{\ \overline{I}\ \boxed{\overline{s}}\ \overline{M}\ \overline{S}\ \}$ *(Object expression)*

$e ::= \cdots \mid \textbf{abstract}$ *(Expression)* $\quad s ::= \cdots \mid (\ell\ \textbf{as}\ \mathsf{self})/x$ *(Substitution)*

$I^o ::= \textbf{inherit object}\ \{\ \overline{M}\ \overline{S}\ \}\ \textbf{as}\ x$ *(Evaluated inherit clause)*

(E-Inherit/Context)

$$\frac{\langle \sigma, e\rangle \rightsquigarrow \langle \sigma', e'\rangle \qquad e = v.m(\overline{v}) \implies e' = \overline{e}; o \qquad e \neq o}{\langle \sigma, \textbf{object}\ \{\ \overline{I^o}\ \textbf{inherit}\ e\ \overline{I}\ \overline{M}\ \overline{S}\ \}\rangle \rightsquigarrow \langle \sigma', \textbf{object}\ \{\ \overline{I^o}\ \textbf{inherit}\ e'\ \overline{I}\ \overline{M}\ \overline{S}\ \}\rangle}$$

(E-Inherit/Multiple)

$$\frac{\begin{array}{c}\overline{\ell\ \text{fresh}} \qquad \overline{m} = \text{names}(\overline{M}, \overline{S}) \qquad \overline{M'} = \overline{[\mathsf{self}.m/m]M} \qquad \overline{\langle \overline{M_f}, \overline{e}\rangle = \text{body}(\overline{S})} \\ \overline{M_\uparrow} = \text{join}(\overline{M'}\ \overline{M_f}) \qquad \overline{m_\downarrow} = \text{names}(\overline{M_\downarrow}, \overline{S_\downarrow}) \qquad \overline{M'_\uparrow} = \text{override}(\overline{M_\uparrow}, \overline{m_\downarrow}) \qquad \overline{\overline{m}\ \text{unique}}\end{array}}{\begin{array}{l}\langle \sigma, \textbf{object}\ \{\ \overline{\textbf{inherit object}\ \{\ \overline{M}\ \overline{S}\ \}\ \textbf{as}\ x\ \overline{s}\ \overline{M_\downarrow}\ \overline{S_\downarrow}}\ \}\rangle \rightsquigarrow \langle \sigma(\overline{\ell \mapsto \langle \varnothing, \overline{M'}\ \overline{M_f}\rangle}),\\ \quad \textbf{object}\ \{\ \overline{M'_\uparrow}\ \overline{[s][(\ell\ \textbf{as}\ \mathsf{self})/x]M_\downarrow}\ \overline{[\mathsf{self}.m/m]e}\ \overline{[s][(\ell\ \textbf{as}\ \mathsf{self})/x]S_\downarrow}\ \}\rangle\end{array}}$$

Auxiliary Definitions

$$\text{join}(\varnothing) = \varnothing$$

$$\text{join}(\textbf{method}\ m(\overline{x})\ \{\ e\ \}\ \overline{M}) = \begin{cases} m \notin \text{names}(\overline{M}, \varnothing) & \textbf{method}\ m(\overline{x})\ \{\ e\ \}\ \text{join}(\overline{M}) \\ e \equiv \textbf{abstract} & \text{join}(\overline{M}) \\ \textbf{method}\ m(\overline{y})\ \{\ \textbf{abstract}\ \} \in \overline{M} & \text{join}(\overline{M}\ \textbf{method}\ m(\overline{x})\ \{\ e\ \}) \\ \text{otherwise} & \textbf{method}\ m\ \{\ \textbf{abstract}\ \}\ \text{join}(\text{override}(\overline{M}, m)) \end{cases}$$

Figure 11 Multiple uniform identity modification.

accessed through that name within the object. An object contains at most one method with any given name, and there are no part-objects.

Multiple **inherit** statements can appear in an object, treated symmetrically. If the same name is inherited from multiple parents, all but one must be **exclude**d, or a local overriding method declared, if a concrete implementation of that method is to appear in the object. All inheritance expressions are evaluated and the final method set of the object assembled before any initialisation code executes. Initialisation occurs from top to bottom (depth-first search) within each branch of the inheritance hierarchy. Methods are decoupled from their names because aliases may provide multiple equivalent names all reaching the method, while exclusion means that local definitions may not be implemented in the final object.

If amelia wished to simultaneously be a graphic and a gunslinger, then two inherit clauses can be included under the transformation model. If the gunslinger class also contained a draw method for drawing her gun, amelia would be required to choose a single implementation by excluding one of the two. The excluded method can still be accessed if it is aliased:

```
def amelia = object {
    inherit gunslinger
    inherit graphic alias draw as render exclude draw
    def image = images.amelia
    var name := "Amelia"
}
```

Even if the gunslinger class also has a name method, amelia has successfully combined the two by overriding both. Note that method resolution is consistent throughout the entire object:

Extended Syntax

$I \quad ::= \quad \textsf{inherit}\ e\ \textsf{alias}\ \overline{m\ \textsf{as}\ m}\ \textsf{exclude}\ \overline{m} \qquad \textit{(Inherit clause)}$

$$\frac{\begin{array}{c}\overline{\ell\ \text{fresh}} \qquad \overline{\overline{m} = \text{names}(\overline{M}, \overline{S})} \qquad \overline{\overline{M'} = \overline{[\textsf{self}.m/m]M}} \qquad \overline{\langle \overline{M_f}, \overline{e}\rangle = \text{body}(\overline{S})} \\ \overline{\overline{M_a} = \text{aliases}(\overline{m_a\ \textsf{as}\ m'_a}, \overline{M'\ M_f})} \qquad \overline{\overline{M_e} = \text{excludes}(\overline{m_e}, \overline{M_a})} \qquad \overline{M_\uparrow} = \text{join}(\overline{\overline{M_e}}) \\ \overline{\overline{m_\downarrow} = \text{names}(\overline{M_\downarrow}, \overline{S_\downarrow})} \qquad \overline{M'_\uparrow} = \text{override}(\overline{M_\uparrow}, \overline{m_\downarrow})\end{array}}{\begin{array}{c}\langle \sigma, \textsf{object}\ \{\ \overline{\textsf{inherit object}\ \{\ \overline{M}\ \overline{S}\ \}\ \textsf{alias}\ \overline{m_a\ \textsf{as}\ m'_a}\ \textsf{exclude}\ \overline{m_e}}\ \overline{s}\ \overline{M_\downarrow}\ \overline{S_\downarrow}\ \}\rangle \rightsquigarrow \\ \langle \sigma, \textsf{object}\ \{\ \overline{M'_\uparrow}\ \overline{[s]M_\downarrow}\ \overline{\overline{[\textsf{self}.m/m]e}}\ \overline{[s]S_\downarrow}\ \}\rangle\end{array}}\ \overline{\overline{m}\ \text{unique}}$$

Auxiliary Definitions

$$\text{aliases}(\varnothing, \overline{M}) = \overline{M} \qquad \text{aliases}(m\ \textsf{as}\ m'\ \overline{m\ \textsf{as}\ m'}, \overline{M}) = \text{aliases}(\overline{m\ \textsf{as}\ m'}, \text{alias}(\overline{M}, m\ \textsf{as}\ m'))$$
$$\text{excludes}(\varnothing, \overline{M}) = \overline{M} \qquad \text{excludes}(m\ \overline{m}, \overline{M}) = \text{excludes}(\overline{m}, \text{exclude}(\overline{M}, m))$$

$$\begin{aligned}\text{alias}(\varnothing, m\ \textsf{as}\ m') &= \varnothing \\ \text{alias}(\textsf{method}\ m(\overline{x})\ \{\, e\,\}\ \overline{M}, m\ \textsf{as}\ m') &= \textsf{method}\ m(\overline{x})\ \{\, e\,\}\ \textsf{method}\ m'(\overline{x})\ \{\, e\,\}\ \text{alias}(\overline{M}, m\ \textsf{as}\ m') \\ \text{alias}(M\ \overline{M}, m\ \textsf{as}\ m') &= M\ \text{alias}(\overline{M}, m\ \textsf{as}\ m')\end{aligned}$$

$$\begin{aligned}\text{exclude}(\varnothing, m) &= \varnothing \\ \text{exclude}(\textsf{method}\ m(\overline{x})\ \{\, e\,\}\ \overline{M}, m) &= \textsf{method}\ m\ \{\, \textsf{abstract}\,\}\ \text{exclude}(\overline{M}, m) \\ \text{exclude}(M\ \overline{M}, m) &= M\ \text{exclude}(\overline{M}, m)\end{aligned}$$

Figure 12 Method transformation modification, with context rule omitted.

the local request to draw in the graphic initialisation will unholster instead of rendering.

The implementation of method transformation multiple uniform inheritance is given in Figure 12 as a modification of the previous uniform multiple inheritance extension. Inherit clauses no longer have super-names, using the method aliasing and excluding syntax instead. The obvious modifications to the context rule to account for method transformations instead of super-names are omitted.

Rule E-Inherit/Transform pre-processes the methods of each inherited object before passing them to join, with the aliases and excludes auxiliary functions. Both of these functions proceeds as expected: fold the transformations over the methods, applying the alias or exclude rules in order. Because these rules are ordered, it is possible to create an alias of an existing alias that occurs earlier in the list, and it is possible to exclude aliases. Exclusion is always processed after aliases, so a directly excluded method cannot be aliased.

Uniform method transformation permits both down-calls and registration. It provides stability through time, but not through local analysis of visible declarations. It imposes the same freshness requirements as the other models, and supports multiple inheritance. Applying method transformation to the simpler object inheritance models continues to maintain their own particular properties, as they were never stable to begin with.

6.3 Positional

The previous multiple inheritance models treated the inherit clauses as symmetric, such that no definition in any one was preferred in the case of conflicts, requiring resolution either by

overriding or method transformations. Moreover, they did not permit any initialisation until all of the direct inheritance was completed. This can be a problem if, for instance, fields need to be initialised for down-calls in super-initialisation code, as with amelia's image field. Positional inheritance addresses these concerns, at the cost of visibly mutating the object during construction, similarly to what can be achieved with mutable object structure.

Under positional inheritance, multiple **inherit** clauses can appear in an object, amongst the typical statements. These inheritances are processed imperatively where they appear, in order, using the same semantics as the selected base model of inheritance. As under multiple uniform, each **inherit** statement can have a name associated. Positional could also be used for single inheritance, allowing some initialisation before the super-constructor is called, but the visible mutation is still present, and the distinct ordering of the inherit clauses in an object body implies a natural method conflict resolution.

In the most general version, an **inherit** clause can appear anywhere in the object body, and have other code before, after, and in between, with the semantics of the inheritance taking effect at the point of appearance and later parents having precedence over earlier. This allows some interesting programmer choices with some of the base models. Altering the order of parents affects which versions of same-named methods are accessed, and interleaving other code in between exposes both at different times. The availability and safety of upcalls and down-calls are affected by the placement of the inheritance, field initialisation, and other code. A more restrained approach could limit inheritance to appearing all at the top (or at the bottom) of the object body.

Positional delegation with named **super**s is essentially the behaviour of Self [10], where multiple parent pointers may exist in a single object; Self does not allow initialisation code to execute in the context of the object under construction or have any priority between parents, but does allow parent pointers to be mutable. The nature of concatenation fundamentally supports positional multiple inheritance, simply copying in the contents of the inherited object in place of the **inherit** statement, and the limitation in the base model was purely syntactic. Named **super**s (or a next-method functionality) are also necessary to access overridden methods. Multiple forwarding is quite straightforward, and strictly named **super**s are not required: because forwarding only accesses the public interface, an ordinary reference to each parent suffices. One of the primary use-cases of positional inheritance — initialising fields before invoking a super-constructor — is irrelevant without uniform identity, as the super-constructor cannot make a down-call into the inheriting object anyway.

Merged identity does not lend itself to the positional extension because it relies on taking over the identity of the parent object, which with multiple parents would result in repeated identity changes, some of which may even lose methods. A different extension could merge multiple identities together, or resolve the resulting issues in some other way, but we do not address this combination here and simply exclude it from consideration as confusing at best.

Positional inheritance is the only one of our multiple-inheritance models that permits inheriting from something obtained from a parent. A parent could define a number of specialised "inner classes", with the intention that its child would in turn inherit from one of those specialisations as well. It is not obvious to us that such an ability is useful, but nor is it obvious that it is not. We note this unique ability, but do not pursue it further. The most interesting version of positional inheritance is based on uniform identity, so we will focus on that extension for the remainder of this section. Each **inherit** is processed in order of appearance, using the same structure-then-initialisation process as under uniform identity. Again, a single identity exists for all initialisation, and again down-calls are valid throughout.

Extended Syntax

$e ::= \cdots \mid \mathsf{super}\ \mathbf{inherit}\ e\ \mathbf{as}\ x\ \overline{s} \mid \overline{i}\ \mathbf{inherit}\ e\ \mathbf{as}\ x\ \overline{s}$ *(Expression)*

$S ::= \cdots \mid \mathbf{inherit}\ e\ \mathbf{as}\ x$ *(Statement)*

$r ::= \cdots \mid (\ell\ \mathbf{as}\ \ell)$ *(Receiver)*

$o ::= \mathbf{object}\ \{\ \overline{s}\ \ \overline{M}\ \ \overline{S}\ \}$ *(Object expression)*

$i ::= \langle \ell, \overline{M}, \overline{s} \rangle$ *(Inherit context)*

$s ::= \cdots \mid (\ell\ \mathbf{as}\ \ell)/x \mid \overline{i}/\mathsf{super}$ *(Substitution)*

(E-Object/Positional)

$$\frac{\ell\ \text{fresh} \qquad \overline{m} = \text{names}(\overline{M}, \overline{S}) \qquad \langle \overline{M_f}, \overline{e} \rangle = \text{body}(\overline{S})}{\langle \sigma, \mathbf{object}\ \{\ \overline{s}\ \ \overline{M}\ \ \overline{S}\ \} \rangle \rightsquigarrow \langle \sigma(\ell \mapsto \langle \varnothing, \overline{[s][\mathsf{self}.m/m]M}\ \ \overline{M_f} \rangle), \overline{[s]}[\langle \ell, \overline{M}\ \ \overline{M_f}, \overline{s} \rangle / \mathsf{super}][\ell/\mathsf{self}]\overline{[\mathsf{self}.m/m]e};\ \ell \rangle}\ \overline{m}\ \text{unique}$$

(E-Inherit/Positional)

$$\frac{\begin{array}{c}\ell\ \text{fresh} \qquad \overline{m} = \text{names}(\overline{M}, \overline{S}) \qquad \overline{M_\uparrow} = \overline{[s_\uparrow][\mathsf{self}.m/m]M} \\ \langle \overline{M_f}, \overline{e_\uparrow} \rangle = \text{body}(\overline{S}) \qquad \ell_\downarrow = \text{first}(\text{last}(i)) \qquad i' = \text{add-subst}((\ell\ \mathbf{as}\ \ell_\downarrow)/x, \overline{i})\end{array}}{\begin{array}{l}\langle \sigma, \overline{i}\ \mathbf{inherit\ object}\ \{\ \overline{s_\uparrow}\ \ \overline{M}\ \ \overline{S}\ \}\ \mathbf{as}\ x\ \overline{s}; e \rangle \rightsquigarrow \langle \text{update}(\sigma(\ell \mapsto \langle \varnothing, \overline{M_\uparrow}\ \ \overline{M_f} \rangle), \overline{M_\uparrow}\ \ \overline{M_f}, \overline{i'}), \\ \overline{[s_\uparrow]}[\langle \ell, \overline{M}\ \ \overline{M_f}, \overline{s_\uparrow} \rangle\ \overline{i'}/\mathsf{super}][\ell_\downarrow/\mathsf{self}]\overline{[\mathsf{self}.m/m]e_\uparrow};\ \overline{[s]}[\mathsf{self}.m/m][i'/\mathsf{super}][(\ell\ \mathbf{as}\ \ell_\downarrow)/x]e \rangle\end{array}}\ \overline{m}\ \text{unique}$$

Extended Auxiliary Definitions

$$\text{body}(\mathbf{inherit}\ e\ \mathbf{as}\ x\ \overline{S}) = \langle \overline{M}, \mathsf{super}\ \mathbf{inherit}\ e\ \mathbf{as}\ x\ \overline{e} \rangle \qquad \mathbf{where}\ \langle \overline{M}, \overline{e} \rangle = \text{body}(\overline{S})$$

$$\text{add-subst}(s, \langle \ell, \overline{M}, \overline{s} \rangle\ \overline{i}) = \langle \ell, \overline{M}, \overline{s}\ s \rangle\ \overline{i}$$

$$\text{update}(\sigma, \overline{M_\uparrow}, \varnothing) = \sigma$$

$$\text{update}(\sigma, \overline{M_\uparrow}, \langle \ell, \overline{M}, \overline{s} \rangle\ \overline{i}) = \text{update}(\sigma(\ell \mapsto \langle \overline{F}, \overline{M'_\uparrow}\ \ \overline{M'}\ \ \overline{M'_\downarrow} \rangle), \overline{M'_\uparrow}\ \ \overline{M'}, \overline{i})$$

$\mathbf{where}\ \overline{m} = \text{names}(\overline{M}, \varnothing)\ \mathbf{and}\ \overline{m_\uparrow} = \text{names}(\overline{M_\uparrow}, \varnothing)\ \mathbf{and}\ \overline{M'_\uparrow} = \text{override}(\overline{M_\uparrow}, \overline{m})\ \mathbf{and}\ \langle \overline{F}, \overline{M_\downarrow} \rangle = \sigma(\ell)$
$\mathbf{and}\ \overline{m_\downarrow} = \text{names}(\overline{M_\downarrow}, \varnothing)\ \mathbf{and}\ \overline{M'} = \overline{[s][\mathsf{self}.m_\uparrow/m_\uparrow][\mathsf{self}.m_\downarrow/m_\downarrow]M}\ \mathbf{and}\ \overline{M'_\downarrow} = \text{override}(\overline{M_\downarrow}, \overline{m}\ \overline{m_\uparrow})$

Figure 13 Positional uniform identity modification, with context rule omitted.

Positional inheritance reintroduces mutation during construction to uniform identity, because each **inherit** adds new methods to the object. When multiple methods are inherited by the same name, the last-inherited method wins out. An unusual aspect is that while down-calls are always available, during construction 'side-calls' to co-parents of a common child can be made only to parents whose **inherit** preceded this one. An object can even define a fresh constructor directly inside of itself, and then inherit from it.

Each line of initialisation occurs after preceding inheritance statements and before subsequent inheritance statements. If **inherit** precedes a field initialisation or other expression, upcalls to that parent are available from that expression; if **inherit** follows a field initialisation, down-calls from that parent accessing that field are safe.

The implementation of positional uniform identity inheritance is provided in Figure 13 as an extension to the core Graceless language; the rule for fresh inheritance is omitted again. The primary difficulty with implementing positional inheritance is that the meaning of local definitions can change imperatively: a local request might refer to some definition in the surrounding scope, but after processing an inherit clause during the initialisation

phase that request might now refer to an inherited definition instead. This presents a particular difficulty for substitution, which irreversibly binds an unqualified name to a particular definition. Substitutions are now delayed by all object expressions, as the inherit clauses are now nested in the object statements. Rule E-Object/Positional replaces the Rule E-Object, handling the new statement form with the updated body translation and applying the delayed substitution.

In order to implement the dynamic scoping, we introduce an *inherit context* i, which records the reference ℓ of the object that an inherit clause appeared inside, the source of the methods $\overline{M}$ that appeared directly in that object, and the delayed substitutions $\overline{s}$ that were on that object. Any processed inherit clause has an ordered stack of these contexts on it, from the actual object the inherit clause appeared in, down to the bottom-most inheriting object. The super prefix is used as a placeholder on newly created inherit expressions, so that Rule E-Object/Positional can substitute it out for the initial inherit context.

By retaining the source of the methods and the substitution scope they appeared in, the methods can be re-substituted in any new scope that appears. The update auxiliary function applies this for any newly inherited methods $\overline{M_{\uparrow}}$ to every object in the current stack of contexts. Rule E-Inherit/Positional handles any inherit expression, constructing a new part-object ℓ, and using update to include the new methods in the original object and every intervening part-object, after applying overrides from the existing methods. The value of self is bound in the inherited object body by the last object reference in the inherit context, as that is the location of the original object. The inherited body is processed in the same way as in Rule E-Object/Positional. One of the key distinctions here is that the value of self to substitute in the inherited body already exists, whereas under single uniform identity the body is concatenated into an object expression and self is substituted once that is evaluated.

Note that inherit expressions still delay substitutions, preventing them from applying to expressions later in any sequence. After each inherit expression is evaluated, the substitutions are then applied to the following expressions, after being shadowed by inherited definitions and the super-name defined by the **as** clause. Positional uniform inheritance preserves the other traits of uniform identity from Section 5.2, with the exception of stability: during construction, an object's apparent structure and behaviour can change. Applied to the simpler object inheritance models, it preserves each of their unique properties.

7 Discussion

Table 1 compares the models according to the criteria established in Section 2. Each model provides a different mix of the criteria, which may be appropriate for different circumstances or languages. The uniform identity design provides the closest match to Java semantics (given at the bottom of the table) while the other multiple-inheritance models following trade off one or more of these properties. No model provides every property; indeed, stability, downcalls, and inheriting from existing objects are fundamentally in conflict. As well, the complexity of each design and its implementation roughly increases down the table, which is a further trade-off for language designers to consider.

While delegation, forwarding, and concatenation can fundamentally support inheriting from arbitrary objects, the other models lean towards supporting planned reuse rather than ad-hoc reuse — that is, inheriting from objects that have been designed to be inherited from, rather than from any arbitrary object. Both planned and unplanned reuse have solid software-engineering motivations; indeed, language features exist both specifically to prevent inheritance (final or sealed classes) and to enable ad-hoc reuse (structural types).

Table 1 Comparison of models of object-first inheritance. * indicates answer holds during construction, but is reversed after. Overload indicates multiple definitions of the same method name in an object, accessible through super-references. Parent indicates ability to inherit from something obtained through another parent. All other columns relate to criteria from Section 2.

	Reg.	Down.	Dist.	Stable	Exist.	Mult.	Overl.	Par.
Forwarding	no	no	yes	yes	yes	no	yes	no
Delegation	no	no*	yes	no	yes	no	yes	no
Concatenation	no	no*	no	no	yes	no	yes	no
Merged	yes	no*	no	no*	fresh	no	yes	no
Uniform	yes	yes	no	yes	fresh	no	yes	no
Mult. Uniform	yes	yes	no	yes	fresh	yes	yes	no
Transform U.	yes	yes	no	no	fresh	yes	no	no
Positional U.	yes	yes	no	no	fresh	yes	yes	yes
Java	yes	yes	no	yes	class	no	yes	no

We do not wish to present one or another choice as better, but to draw attention to a potentially-unintended side effect of various points in the solution space. Nonetheless, it is possible for any of the fresh-object–based systems to support delegation or forwarding semantics simply by exposing a method, accepting any object as an argument, that returns a fresh object whose methods provide the behaviour in question. Concatenation semantics can similarly be supported by inheriting from a standard clone.

Diamond inheritance (repeatedly inheriting from the same class or trait two or more times) has long been recognised as a problem in object-oriented language design. Eiffel and C++ both offer the same essential solution to the problem: arranging that some classes can be replicated each time they are inherited, while other classes will be inherited only once. Malayeri and Aldrich [36] present a good discussion of the problems diamonds cause for inheritance, and then argue that diamond inheritance can be prevented in languages, partly by supporting a *requires* clause inspired by Scala which indicates that a trait depends upon the eventual final **self** object providing a set of methods, but not actually implementing those methods itself. The two multiple-inheritance systems we describe are open to this sort of collaboration, but do not require it. Because they are object-based, rather than class-based, some issues of diamond inheritance do not arise, as each instance of a parent is (unavoidably) separately obtained and constructed. In an object-based system, coalescing similar ancestors is a dubious activity, as side effects may occur on the path to construction and be semantically meaningful, which cannot happen in a static, declarative class system.

In our formal model we have made a conscious effort to handle as many errors as possible in the operational semantics (i.e., at run time) rather than by defining erroneous programs as ill-formed (i.e., at compile time). There are several reasons for this, but most important is that we see further layers on top — such as type systems or checks for diamond inheritance — as an important, but separable part of the design process. Omitting such definitions highlights the inheritance designs, and enables the core language of the model to be smaller and more general. In fact, a language could provide the ability to impose additional families of static restrictions at the surface level [23].

Similarly, we take a pragmatic approach to object initialisation: access to uninitialised variables raises a runtime error. This choice has helped simplify our inheritance designs, as we have not needed to contort the models to ensure that e.g. all variables are definitively initialised. Just as a range of static type systems can be layered over the language's semantics, so we expect that a safe initialisation scheme, such as Delayed Types [16], Masked Types [42],

Hard Hats [20, 54], Freedom Before Commitment [47], or the Billion Dollar Fix [43] should be able to to be layered on top of the initialisation semantics in the model. Highlighting where such runtime errors are liable to occur in the different designs helps language designers to choose where to trade off for or against additional safety.

8 Related Work

Class-based object-oriented languages begin with SIMULA [2], and much of the conceptual framework of object-orientation descends from the Scandinavian school founded by Dahl and Nygaard, viewing programming as simulation, and consequently programs as models of phenomena in the real world [35]. Taivalsaari argues the class-based understanding of programming is also 'classical' in the sense of descending from the classical philosophy of Plato and Aristotle [49].

Simula's model of objects as instances of classes was greatly expanded on by Smalltalk [4]: unlike SIMULA, Smalltalk classes are also instances of other (meta-) classes themselves, importing the power (and also the complexity) of Lisp-style computational reflection into object-oriented languages [45]. Lisp then returned the favour with a series of object-oriented extensions culminating in the CLOS Meta-Object Protocol [29]. We also owe the notion of 'static' (per class) declarations, distinct from per-instance declarations, to Smalltalk.

The complexity of Smalltalk's meta-model clearly inspired Lieberman to propose languages based purely on objects, with delegation as the sharing mechanism [32], which then led to a general interest in 'prototype-based' programming languages, i.e. languages, following Self [51] that create new instances by copying existing instances, rather than by an apostrophe to a class. Emerald [3] also lacked classes, but objects were created by literal expressions.

The fates of Emerald and Self are instructive. By 1991 Emerald had a *"syntactic construct called a class that provides the functionality normally expected of classes"* [24]. At the same time, Self's programming style was based on separate objects corresponding to instance prototypes (defining all the instance variables in one object) and method suits (called "traits") with each conceptual class giving rise to both a trait, and a prototype delegating to that trait; traits were also linked by delegation corresponding to the conceptual class inheritance [50]. By 1995, Self objects were effectively given a single "copy down parent" attribute, and slots from that object would be copied into the object whenever it was edited. This is how e.g. if an extra 'z' slot was added into the graphic object, it would also be added into the gRectangle object. Eventually, a "Subclass me" button was added to the IDE, which copied both instance object and trait objects, recreating the class-like structure with delegation and copy-down parent links configured correctly.

Dony et al. modelled a range of different designs for class-, object-, and prototype- based languages, although by writing a software product line of definitional interpreters in Smalltalk, rather than modelling language features formally [13, 12]. Taivalsaari et al. [40] surveyed other contemporaneous research along these lines.

Relatively early on, Eiffel incorporated a sophisticated class-based multiple inheritance with deep renaming (rather than shallow aliasing) and exclusion ('undefine') and repeated inheritance [38]. Eiffel's design has many advantages, notably that an object implementing several different types can have multiple implementations of methods with the same name, depending on the type in which those definitions originate. Eiffel's development environment can generate the 'flat form' of the class: unfortunately because of Eiffel's type-aware semantics, the flattened form cannot represent all the possible behaviours of the class. Over the years, other languages incorporated many of the features of Eiffel's design, notably C++.

The other main stream of work is based on mixins, rather than classes [7]. Unlike CLOS or Eiffel multiple superclasses, mixins are applied one at a time, in a linear order specified by the programmer. Bracha's Jigsaw formalised mixin composition in a class-based style, along with a rich trait algebra including merge, restrict, select, project, overriding, and rename operators [8, 5]. Flatt et al. [18, 19] develop a semantics for classes and class-like mixins (without composition operators) in a core language, and have incorporated mixins and traits into Scheme (now Racket) based on classes and macros. Lagorio et al. [30] modeled Jigsaw in a class-based formalism based on Featherweight Java [26], and then Corradi et al. [11] extended the formalism to handle family polymorphism [15]. More recently, class mixins have been incorporated into Newspeak (alongside family polymorphism) [9] and Dart [6].

Traits were revived for multiple inheritance in Smalltalk [14]: in this design, a class can inherit from one single superclass, and then incorporate multiple traits based with a trait algebra comprising sum, aliasing, exclusion operators. A key property of Smalltalk traits is that their conceptual model was based on flattening, rather than dynamic dispatch, although both semantics have been proven equivalent [39]. Scala [41] and Java 8 [22] incorporate traits, although relying on 'super' calls rather than aliasing or renaming.

A discussion of object-based inheritance systems would be incomplete without referring to OCaml [31], which is not dissimilar to the language of our base model. Both languages have object constructors, classes as sugar for methods returning fresh objects, symmetric multiple inheritance, and are structurally typed. We work on the theory of traditional object-style polymorphism rather than OCaml's row polymorphism.

More significantly, even if OCaml classes are described as syntactic sugar, objects (or other classes) can only inherit from classes, whereas in all of our models objects can inherit from (at least) any manifestly fresh objects whether or not defined by classes. OCaml also has a more complex, but in some ways less powerful, initialisation model than what we specify here. In OCaml, field initialisers are evaluated in the enclosing lexical scope, and initialisation code must be sequestered into initialisation blocks which are run late — neither of which reflect a straightforward reading of program's source code — while in our models initialisation is always in a straight line, within the context of the object being constructed.

9 Conclusion

Object-based inheritance is unexpectedly complicated, especially when commonplace desires for functionality available in classical models are involved, and programmers have resorted to increasingly complex workarounds in existing object-based languages. In this paper we showed that object inheritance without classes is both viable and desirable, avoiding the conceptual complexity of an additional kind of entity in an object-oriented language without losing functionality through careful feature selection, and set out a range of options with their various trade-offs made explicit.

We presented and discussed seven models of object inheritance, including the well-known approaches of delegation, forwarding, and concatenation. We presented a novel extended operational semantics for a base language incorporating advanced but standard features affected by inheritance, and formalised the models as extensions to that single base language, formally demonstrating the subtle behavioural differences of each model. In particular, we addressed the complex questions of downcalls, object registration, stability, inheriting preëxisting objects, action at a distance, and multiple inheritance, and their interactions, illustrating that object-based inheritance has the full range of possibilities of classical inheritance, and showed that many of these models can be used as effectively as purely declarative classes, but particular combinations — especially class initialisation semantics

combined with inheritance from preëxisting objects — require a specific set of features usually reserved for very dynamic and reflective languages.

References

1 Henk Barendregt. *The Lambda Calculus*. North-Holland, revised edition, 1984.
2 Graham M. Birtwistle, Kristen Nygaard, Bjørn Myhrhaug, and Ole-Johan Dahl. *Simula Begin*. Studentlitteratur, 1979.
3 Andrew P. Black, Eric Jul, Norman Hutchinson, and Henry M. Levy. The development of the Emerald programming language. In *History of Programming Languages III*, 2007.
4 Alan Borning. Classes versus prototypes in object-oriented languages. In *Proc. ACM Fall Joint Computer Conference*, pages 36–40, 1986.
5 Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modules, and Multiple Inheritance*. PhD thesis, University of Utah, 1992.
6 Gilad Bracha. Mixins in Dart, 2015. [Online; accessed 30-November-2015]. URL: `https://www.dartlang.org/articles/mixins`.
7 Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA*, 1990.
8 Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proc. International Conference on Computer Languages*, pages 282–290, 1992.
9 Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in Newspeak. In *ECOOP*, pages 405–428, 2010.
10 Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are shared parts of objects: inheritance and encapsulation in Self. *Lisp and Symbolic Computation*, 1991.
11 Andrea Corradi, Marco Servetto, and Elena Zucca. DeepFJig: modular composition of nested classes. In *PPPJ*, pages 101–110, 2011.
12 Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based languages: From a new taxonomy to constructive proposals and their validation. In *OOPSLA*, 1992.
13 Christophe Dony, Jacques Malenfant, and Pierre Cointe. Classifying prototype-based programming languages. In James Noble, Antero Taivalsaari, and Ivan Moore, editors, *Prototype-Based Programming: Concepts, Languages and Applications*, chapter 2. 1999.
14 Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 2005.
15 Erik Ernst. Family polymorphism. In *ECOOP*, 2001.
16 Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In *OOPSLA*, pages 337–350, 2007.
17 Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.
18 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL*, 1998.
19 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, 1998.
20 Joseph (Yossi) Gil and Tali Shragai. Are we ready for a safer construction environment? In *ECOOP*, pages 495–519, 2009.
21 Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
22 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Oracle, 2015.
23 Michael Homer, Timothy Jones, James Noble, Kim B. Bruce, and Andrew P. Black. Graceful dialects. In *ECOOP*, pages 131–156, 2014.

24 Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald programming language report. Computer Science, UBC, October 1991.

25 Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *History of Programming Languages III*, 2007.

26 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

27 Carlton Egremont III. *Mr. Bunny's Big Cup o'Java*. Addison-Wesley, 1999.

28 Timothy Jones and James Noble. Tinygrace: A simple, safe and structurally typed language. In *Formal Techniques for Java-like Programs*. 2014.

29 Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

30 Giovanni Lagorio, Marco Servetto, and Elena Zucca. Featherweight Jigsaw: Replacing inheritance by composition in Java-like languages. *Inf. Comput.*, 214:86–111, 2012.

31 Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.00 documentation and user's manual, 2012.

32 Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *OOPSLA*, November 1986.

33 Henry Lieberman, Lynn Andrea Stein, and David Ungar. Treaty of Orlando. In *Addendum to OOPSLA Proceedings*, May 1988.

34 Lua-Users. Object oriented programming, 2014. [Online; accessed 30-November-2015]. URL: `http://lua-users.org/wiki/ObjectOrientedProgramming`.

35 Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

36 Donna Malayeri and Jonathan Aldrich. Cz: Multiple inheritance without diamonds. In *OOPSLA*, 2009.

37 Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

38 Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

39 Oscar Nierstrasz, Stéphane Ducasse, and Nathanael Schärli. Flattening traits. *Journal of Object Technology*, 5:66–90, 2006.

40 James Noble, Antero Taivalsaari, and Ivan Moore, editors. *Prototype-Based Programming: Concepts, Languages, Applications*. Springer-Verlag, 1997.

41 Martin Odersky, Lex Spoon, and Bill Venners. *Programming In Scala*. Artima, 2011.

42 Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In *POPL*, pages 53–65, 2009.

43 Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. The billion-dollar fix—safe modular circular initialisation with placeholders and placeholder types. In *ECOOP*, pages 205–229, 2013.

44 Pat Shaughnessy. *Ruby Under A Microscope*. No Starch Press, 2013.

45 Brian C. Smith. Reflection and semantics in Lisp. In *Proceedings of the ACM Conference on Principles of Programming Languages*, 1984.

46 Stack Overflow. Which JavaScript library has the most comprehensive class inheritance support?, 2015. [Online; accessed 30-November-2015]. URL: `https://stackoverflow.com/questions/711209`.

47 Alex J. Summers and Peter Müller. Freedom before commitment - a lightweight type system for object initialisation. In *OOPSLA*, pages 1013–1032, 2011.

48 Antero Taivalsaari. Delegation versus concatenation or cloning is inheritance too. *Oops Messenger*, 6(3), 1995.

49 Antero Taivalsaari. Classes vs. prototypes: Some philosophical and historical observations. In James Noble, Antero Taivalsaari, and Ivan Moore, editors, *Prototype-Based Programming: Concepts, Languages and Applications*, chapter 1. Springer-Verlag, 1999.

50 David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp and Symbolic Computation*, 4(3), June 1991.

51 David Ungar and Randall B. Smith. SELF: The power of simplicity. *Lisp and Symbolic Computation*, 4(3), June 1991.

52 Wikipedia. Snit, 2015. [Online; accessed 30-November-2015]. URL: `https://en.wikipedia.org/w/index.php?title=OTcl&oldid=663399454`.

53 Allen Wirfs-Brock, editor. *ECMAScript 2015 Language Specification*. Ecma International, 6th edition, 2015.

54 Yoav Zibin, David Cunningham, Igor Peshansky, and Vijay Saraswat. Object initialization in X10. In *ECOOP*, pages 207–231, 2012.

One Way to Select Many*

Jaakko Järvi[1] and Sean Parent[2]

1 **Texas A&M University**
College Station, TX, USA
jarvi@cse.tamu.edu
2 **Adobe Systems Inc.**
San Jose, CA, USA
sparent@adobe.com

Abstract

Selecting items from a collection is one of the most common tasks users perform with graphical user interfaces. Practically every application supports this task with a selection feature different from that of any other application. Defects are common, especially in manipulating selections of non-adjacent elements, and flexible selection features are often missing when they would clearly be useful. As a consequence, user effort is wasted. The loss of productivity is experienced in small doses, but all computer users are impacted. The undesirable state of support for multi-element selection prevails because the same selection features are redesigned and reimplemented repeatedly. This article seeks to establish common abstractions for multi-selection. It gives generic but precise meanings to selection operations and makes multi-selection reusable; a JavaScript implementation is described. Application vendors benefit because of reduced development effort. Users benefit because correct and consistent multi-selection becomes available in more contexts.

1998 ACM Subject Classification D.2.11 Software Architectures: Domain-specific architectures; D.2.13 Reusable Software: Reusable libraries

Keywords and phrases User interfaces, Multi-selection, JavaScript

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.14

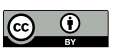
Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.2.1.7

1 Introduction

Many, perhaps most, interactive software applications present their users one or more collections of elements in the form of lists, trees, grids, or otherwise arranged views, of which a user can select one or more elements. Examples include selecting files and folders in a file explorer; mail folders or mail messages in a mail client; music tracks in a media player; thumbnail images in a photograph organizer; "to do" list items, hours, days, weeks, or months in a calendar application; pages organized into "tabs" in a web browser; and electronic books or videos on a digital library or store. These tasks are typical daily activities for many computer users—we select elements from collections dozens of times per day.

Regardless of which set of modern applications a user chooses for mail, music, photos, calendar, web browsing, books, and videos, the features for selecting elements are likely to differ across applications—even within a single application the selection features for different collections, such as the list of mail folders and list of mail messages, are likely to be different.

* This work was supported in part by NSF grants CCF-0845861 and CCF-1320092.



The differences could presumably stem from optimizing the feature for the best possible user experience in different kinds of selection contexts, but this is not the case. The selection features of modern, widely used applications are different in quite arbitrary ways and the results they produce can be unexpected; Section 2 presents examples. A more plausible reason for the variability is that the software industry has not managed to establish precise standard semantics for selecting multiple elements from collections and, consequently, to produce easily reused implementations of selection features. Each development team therefore designs and implements selection features anew. This repeated design and development effort is a strain on development resources, and a cause for what we have today:

- At a cursory level, most applications offer similar multi-element selection capabilities (*multi-selection* for short). The detailed semantics, however, are "artifacts of the implementation"; instead of being designed for the best user experience, they are what falls out of particular implementation choices.
- User interfaces have many contexts where multi-selection would be useful but it is not provided because of the considerable development effort that would be required to do so. To a user there is no good reason why multi-selection cannot be used with browser tabs, applications in the OS X dock, or icons in the "flyout folders" that open from the dock.
- Even when multi-selection is provided, the feature is incomplete. We draw attention to one particular deficiency. Selections are constructed and modified with key-commands and mouse-clicks that are issued almost reflexively. They can grow to become complex objects, but remain brittle. For example, constructing a selection of thumbnail photos in a photo organizer can take minutes—yet at all times the selection is just one mis-click away from vanishing. Almost no applications provide any kind of undo facility for selections.

Many users may consider a less than ideal selection feature a small problem since the inconvenience experienced at any one selection task is usually minor. Due to the omnipresence of selection tasks in all computer use, however, the aggregate effect is not small. Lack of familiar and predictable multi-selection semantics can cause confusion and steer users towards more repetitive and perhaps more frustrating ways to perform tasks, such as repeating "single-element selection followed by a command" interactions multiple times. Frustrating experiences with computer use impact individuals, organizations, and societies negatively [10].

To better the user experience of future software applications, we put forth a proposal for a set of common multi-selection features. We present the justification for the proposed features but do not compare them to other possible feature sets in user studies.

The primary contribution of the paper is not the proposed selection features, which build significantly on well-known guidelines [2, 12], but rather the abstractions that capture the essence of multi-selection features and enable their precise and concise specification, and their generic implementation. We describe a JavaScript library that provides a rich multi-selection feature with support for keyboard and mouse operations, undo/redo commands, and selecting elements based on a predicate. The implementation is not tied to a specific framework or widget. With little per-instance development effort, developers can reuse the feature in different contexts and provide users the same consistent selection semantics everywhere. The paper thus shows how to make a ubiquitous user interface behavior reusable; it replaces a user interface implementation guideline with a precise specification and a reusable implementation.

2 Multi-selection today

This section discusses the selection features of a sample of popular applications. Our intent is not to criticize particular applications, vendors, or developers, but rather to point out the

variation in these features and that even applications with large user bases, after numerous releases, exhibit problems in their selection features.

For a detailed understanding of how multi-selection works in an application, one must resort to reverse engineering through experimenting with the feature. The descriptions in help systems or user guides, if they exist, tend to be cursory. For example, Apple's documentation for OS X El Capitan [4], primarily describing Finder, states the following:

- To move, copy, and make changes to items, you usually have to select them first.
- Select an item: Click the icon for any document, folder, app, or disk.
- Select multiple items: Hold down the Command key, then click the items.
- Select multiple items that are listed together: Click the first item, then press the Shift key and click the last item. All items in between are included in the selection.
- You can also click near the first item, hold down the trackpad or mouse button, then drag over all of the items.
- Select all items in a window: Click a window to make it active, then press Command-A.
- When you selected multiple items and want to deselect one of them, Command-click the item.

This description is vague and incomplete. Aspects not described include selecting multiple disjoint regions with command-click shift-click pairs, drags started with a command-click (they toggle), keyboard selection, the nuances of anchor (explained below) placement, and that shift-click can sometimes deselect. Further, the help document applies only to views that are sequentially ordered; in other views the commands have significantly different meanings.

The above description is of course not intended to be a detailed documentation of Finder's multi-selection features. Such documentation in user guides would be of little benefit—few users today turn to documentation to learn how to use an application, let alone a seemingly trivial feature such as selection. Indeed, users should not need to consult a manual to use multi-selection. Multi-selection gestures and their meaning should be part of a common "user interface language" that we as users know and expect applications to understand.

Pieces that can serve as beginnings of such a language exist. In most applications, a mouse *click* with no modifier keys means selecting an individual element, a *command-click* (on a Mac, *control-click* on a PC) toggling an element, and a *shift-click* selecting elements (from somewhere) "up to" an element. Dragging a mouse to indicate a rectangular area usually means selecting the elements that fall on that rectangle. There are also established key bindings for selecting elements using the keyboard. The precise meanings of these actions, however, are not established, as evidenced by the below review of sample applications.

2.1 OS X Finder multi-selection

Finder supports different kinds of views. A sequential view shows each file as a single line. A two-dimensional view shows files as icons, placed at arbitrary locations. Selection operations are different for different views. We first discuss selection in sequential views. We refer to the objects being selected as *elements*, whether they are files, lines, or icons.

At every click or command-click, Finder establishes an *anchor* at the clicked element. A subsequent shift-click selects the range of elements from the anchor to the clicked element, the *active end* of the range. The terminology comes from the Macintosh guidelines [2]. We refer to the set of elements indicated by the anchor and the active end as the *active (selection) domain*. Further shift-clicks change the active end, and thus also the active domain; a shift-click farther away from the anchor adds elements to the active domain, closer to the anchor removes elements from it, and on the opposite side of the anchor both removes (from

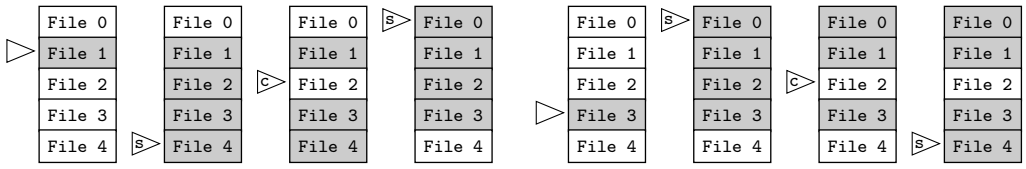


Figure 1 Selection outcomes in Finder. The triangles mark the elements the user clicks: `s` is a shift-click and `c` a command-click. Each column shows the selection state after the click. The darker elements are selected. The series of operations in the diagram (b) is symmetric with that of the diagram (a). The symmetry axis is the element `File 2`. The selection results are not symmetric.

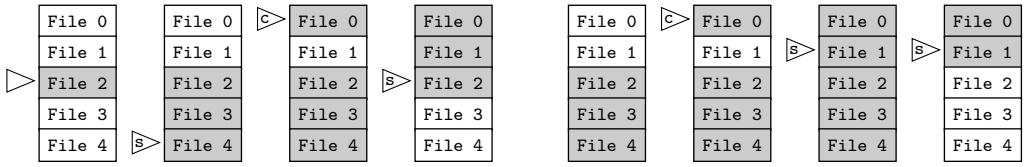


Figure 2 Selection outcomes in Finder. The series (a) shows that when a shift-click occurs on an already selected element, deselections outside the active domain can take place. The series (b) shows that the effect of shift-clicking is not idempotent.

the anchor to the previous active end) and adds (from the anchor to the new active end). Finder does not remember the prior selection state of the elements in the active domain; when a shift-click shrinks the active domain, the elements that were dropped from the domain will unconditionally become deselected, rather than restored to their prior state.

To foresee the effect of a shift-click, the user needs to be aware of the anchor's location. The anchor is not, however, visible to the user, and the rules for where it resides are complex. For example, if a command-click deselects a previously selected element, the anchor is placed on the first selected element downward from that element; or if no such element exists, on the first selected element upward; or if no such element exists, on the first element of the view.

The anchor placement rules lead to surprises, as the two series of operations in Figure 1 demonstrate. It seems reasonable to expect that a command-click followed by a shift-click has no effect outside the range between the locations of these two clicks. Yet in the first series, an element outside of this range toggles from selected to not selected. The second series is by itself not surprising but considered together with the first one it is—two symmetric selection operations lead to asymmetric results.

Another surprising part of the selection rules is the effect of shift-clicking an element that is currently selected: if a shift-click occurs on an element that belongs to a contiguous range of selected elements, all elements in that range are deselected prior to selecting the elements in the new active domain. Figure 2 shows two cases of such overly eager deselection.

It is possible to understand Finder's rules, as our reverse engineering effort demonstrates. We doubt that many will go through a similar exercise. After all, Finder is only one of many contexts of multi-selection and knowing its peculiar set of rules helps little elsewhere.[1]

[1] Finder's multi-selection behavior appears in a few contexts outside of Finder, such as in the File selection dialog shared by many applications and in Apple Mail's (version 6.5) folder and message lists—though not in the list of messages within a thread. Incidentally, the selection features of Apple's iCloud Mail

2.2 Microsoft's File Explorer multi-selection

Microsoft's File Explorer's multi-selection feature is similar, yet subtly different from that of Finder. Click behaves as in Finder: an anchor is established on the clicked element, the element is selected, and all other selections are cleared. Control-click roughly corresponds to command-click on Finder: it toggles the current element and establishes a new anchor. The anchor is always placed on the clicked element, be it selected or not. This is different from Finder, where the anchor is never placed on an unselected element. In addition to setting the anchor, control-click establishes a *mode* of selecting: if the control-click occurred on a selected element, the mode is to deselect, otherwise to select elements. The mode affects subsequent *control-shift-clicks* (Windows' selection documentation does not mention control-shift-click [11]), each of which establishes a new active end and either selects or deselects according to the mode. Shift-click sets the active end, selects the elements in the active domain, and clears all other selections.

File Explorer's model is relatively consistent, but different from most anything else; differences manifest even within File Explorer itself. For instance, changing to a hierarchical view of files disables multi-selection altogether.

2.3 Gmail multi-selection

Gmail's web interface supports multi-selection in the list of mail messages and single-selection in all other collections: mail folders, contacts, and category tabs. Each element on the message list has a checkbox for receiving the mouse clicks that control selection.

Click and command-click are interchangeable: they toggle the element that was clicked and set the anchor to that element. Shift-click sets the active end and either (1) selects the elements from the anchor to the active end, if the active end is on an element that was not selected, or (2) deselects those elements, if the active end is on an element that was selected. Shift-click then places a new anchor, at the active end.

The behavior that results from this rule set differs in at least two significant ways from that of Finder and File Explorer. First, click does not discard the current selection. For that effect, one needs to resort to the keyboard shortcut `*n`, or a click on or above the first selected element followed by a shift-click on the last selected element. Second, whether shift-click selects or deselects elements depends not on the anchor but on the active end; in File Explorer's control-shift-click the anchor's selection status determines the mode of selecting. As a result, in Gmail a shift-click always either grows or shrinks the current selection, not both. The notion of an active domain modified by shift-clicking does not apply.

Gmail's selection rules are relatively simple, but their effect may still surprise. If the current selection is a contiguous region, a user may get the impression that shift-click always grows or shrinks the selection either at the top or bottom of a selected region. If, however, a shift-click that shrinks the region at one end is followed by a shift-click beyond the other end, the region grows at both ends. Figure 3 illustrates such a sequence that arises, for example, in the following scenario: a user (1) clicks to select some message m, (2) shift-clicks to select a few more messages above m, (3) realizes that too many were selected and backs down one or more messages by shift-clicking some message k, still above m, and finally (4) shift-clicks in an attempt to extend the selection with messages below m. That k becomes selected is peculiar, and might go unnoticed as k could reside outside the current viewport.

app, which is made to look quite similar to the desktop mail client, are significantly different.

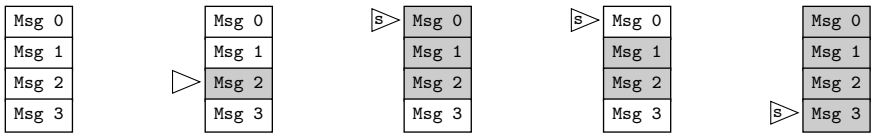


Figure 3 A series of mouse events and their effect in Gmail's multi-selection semantics. That the last shift-click extends the selected region at both ends may surprise.

Gmail exposes the anchor location, which makes it possible to predict the effect of a shift-click in all cases. Gmail also maintains a cursor indicating the current message for keyboard navigation and commands. Clicks move that cursor too, but we have not been able to understand the exact rules guiding those movements.

Gmail's selection behavior is a departure from the conventions in OS X and Windows. The differences manifest concretely when a Gmail user selects files to attach to a message and is subjected to the OS's multi-selection semantics.

2.4 Multi-selection with keyboard

Intuitive keyboard commands for selection are important. The preferred and effective methods to perform selections vary from one user to another. Karat et al. compared touch, mouse, and keyboard in selecting one (menu) element at a time, and concluded that none of the devices was the most effective for or most preferred by all users [8].

File Explorer's keyboard bindings map rather consistently to mouse clicks so that it is possible to use the mouse and the keyboard interchangeably. Space corresponds to a click and arrow keys to a cursor movement followed by a click. The effect of modifiers is the same as for mouse clicks, except that control combined with an arrow is just movement. That control suppresses the automatic click action is useful, so that moving around without altering the current selection is possible. Such capability is missing from, e.g., the MS Word text editor, even though it supports selecting multiple disjoint regions with a mouse.

Finder's keyboard bindings are less complete than File Explorer's. Arrow keys correspond to moving to the neighboring element followed by a click, or shift-click if the shift key is held down. Command-click has no counterpart; keyboard commands thus seem to be insufficiently expressive for selecting disjoint regions.

Gmail's selection key commands are off by default and limited when on: arrow keys move the cursor that indicates the current element, `x` toggles, `*a` selects all, and `*n` deselects all.

2.5 Rubber band selection

A mouse drag can be used to select a range: the anchor is established at the *mousedown* and the active end at the *mouseup* event's position. While a drag is ongoing, the current mouse position is interpreted as the active end. Often user interfaces show the changing minimum bounding rectangle of the points of the anchor and active end during the drag. File Explorer and Finder support this kind of *rubber band selection*, whereas Gmail does not.

In File Explorer, rubber band selection without modifier keys clears the current selection and selects all elements in the specified region; the shift modifier suppresses the clearing of the current selection. Control and shift-control toggle the elements inside the rubber band.

Finder's rubber band selection without modifiers is as in File Explorer, but both the shift and command modifiers toggle elements. No command unconditionally adds elements to or removes them from the current selection. In both OSs, the meaning of modifier keys is different in rubber band selection than in selection through clicking or using the keyboard.

It is curious that in both OSs rubber band selection supports toggling, but not deselecting. Deselecting has obvious uses, such as selecting a range save a few elements, but use cases for a single mouse drag to flip some elements from unselected to selected and others to the opposite direction are hard to come by. Perhaps toggling is implemented to approximate deselection: when the anchor and active end are wholly within an already selected region, toggling and deselecting produce equal results.

Another curious aspect is that the active domain is treated differently when rubber band selecting than when selecting with mouse clicks. In the former case, when the active domain shrinks, elements revealed from under the active domain are unconditionally deselected. In the latter case, however, their selection status prior to their inclusion to the active domain is remembered and restored. Section 4.8 describes the drawbacks of the former behavior.

2.6 Touch applications

In recent years, the popularity of touch devices has risen dramatically. Apple's iOS, Google's Android and Microsoft's Windows 8 are all operating environments designed from the outset for multi-touch displays. Given the long history these companies have with user interfaces, one would have expected the basic concepts, such as selection, to have been refined and carried forward. Instead we find even more disparity and limitations in touch based selection.

In iOS, in Springboard (the application launcher) long-pressing on an icon enters *selection mode*. Only one application can be moved or deleted at a time. In the Mail application, tapping an Edit button (top right of screen) moves into selection mode, where tapping selects messages. After selecting, one chooses an operation to perform on the selection (Trash, Move, Mark All Read, ...). The Photos application works similarly, except that the Edit button is named Select. In the Messages application, again an Edit button (now top left of the screen) enters selection mode. A message is selected by tapping on an indicator to its left. Only one conversation at a time can be selected; the only operation available is Delete.

In Windows 8.1, long-pressing on a title or swiping up from the bottom of the screen and tapping "Customize" on the far right enters selection mode, where multiple tiles can be tapped. In Windows 8.0, selecting a tile was done by swiping down (perpendicular to the horizontal scroll direction). Apparently this approach was abandoned, likely because it was not very discoverable. In the Mail application when one taps on an item, a checkbox appears to its left. Tapping on the check box toggles it on and enters a selection mode where checkboxes appear next to all messages; tapping anywhere on a message toggles its checkbox. Swiping up from the bottom of the screen and tapping "Select" on the left also enters selection mode. In the Photos application a swipe down on a photo toggles its selection state.

In Android 4.4 on the Launcher screen one can only operate on icons individually with a long-press. In the GMail application, one can select multiple email messages by long-pressing on each one individually. In Google Photos one enters selection mode by long-pressing a photo. A drag will then select multiple photos, tap will toggle one photo.

Except for the Google Photos application, none of the above selection models allow quickly selecting a large range of elements. Where multiple selection is supported, each element must be selected one at a time.

3 Selection concepts

The above review of multi-selection features in a handful of well-known applications confirms that the semantics of multi-selection are not well established, and that users are faced with a wide variety of behaviors, some surprising. This section describes such semantics, making no

assumptions about what kinds of elements are being selected, how they are organized and visually presented to the user, and what kind of a selection tool or device the user is using.

The multi-selection task is to select a subset of the elements of a collection presented to the user. For an abstract view of the task, it is not essential what these elements are. It is thus convenient to think of them as an indexed family $x : I \to M$, where M is the set of elements, and I a suitable index set. With this setup, we can talk about the ith element of a collection, where $i \in I$, and mean the element x_i. A set of selected elements in I can be represented as a mapping $s : I \to \{\mathsf{T}, \mathsf{F}\}$, where $s(i) = \mathsf{T}$ means that x_i is selected and $s(i) = \mathsf{F}$ that it is not. We call s a *selection mapping*. Below, we use $\mathbf{2}$ for the set $\{\mathsf{T}, \mathsf{F}\}$.

3.1 Primitive selection operations

A user constructs a selection mapping with a sequence of selection commands, such as a rubber band selection, followed by a shift-click, press of shift-right arrow, and a command-click. Each user command modifies the current selection mapping, and can thus be modeled as a function of type $(I \to \mathbf{2}) \to (I \to \mathbf{2})$. We call functions of this type *selection operations*. We define *primitive* selection operations, from which all selection operations can be built:

▶ **Definition 1.** Let $x : I \to M$ a collection, $J \subseteq I$, and $f : \mathbf{2} \to \mathbf{2}$ a mapping. J and f (uniquely) determine a *primitive selection operation*

$$\mathsf{op}^f_J : (I \to \mathbf{2}) \to (I \to \mathbf{2}), s \mapsto \lambda i. \begin{cases} f(s(i)), & i \in J \\ s(i), & i \notin J \end{cases}$$

In other words, the primitive selection operation op^f_J produces a new selection mapping that applies f to the selection state of every element in J, and has no impact on elements not in J. We call J the *selection domain* and f the *selection function* of op^f_J.

Selection operations compose. Starting from the "no elements selected" selection mapping $e : I \to \mathbf{2}, i \mapsto \mathsf{F}$, the selection that results after applying a series of primitive selection operations $\mathsf{op}^{f_1}_{J_1}, \mathsf{op}^{f_2}_{J_2}, \ldots, \mathsf{op}^{f_n}_{J_n}$ is $(\mathsf{op}^{f_n}_{J_n} \circ \mathsf{op}^{f_{n-1}}_{J_{n-1}} \circ \ldots \circ \mathsf{op}^{f_1}_{J_1})(e)$.

Any change to the selection state of elements can be expressed as a composition of primitive selection operations. The task of implementing a multi-selection feature can thus be viewed as the task of providing the user convenient means to specify primitive selection operations. Since they are uniquely determined by the selection function f and domain J, it is these two components that must be derived from user interactions. We first review common ways in which user actions give rise to particular selection functions.

There are four functions of type $(\mathbf{2} \to \mathbf{2})$, and thus four selection functions: $\lambda x.x$, $\lambda x.\,\mathsf{T}$, $\lambda x.\,\mathsf{F}$, and $\lambda x.\neg x$. Primitive selection operations of the form $\mathsf{op}^{\lambda x.x}_J$ are identity functions. Operations of the form $\mathsf{op}^{\lambda x.\,\mathsf{T}}_J$ select all elements in the selection domain, and arise as a result of many actions, including clicking an element, shift-clicking, or rubber band selection. Operations of the form $\mathsf{op}^{\lambda x.\,\mathsf{F}}_J$ deselect all elements in the selection domain. They can model, e.g., File Explorer's or Gmail's "deselect a region" operation. Operations of the form $\mathsf{op}^{\lambda x.\neg x}_J$ toggle the selection status of the elements in J. Finder's command-click on some element i could be modeled as $\mathsf{op}^{\lambda x.\neg x}_{\{i\}}$. Toggling is often limited to singleton sets, but as discussed in Section 2.5, Finder's and File Explorer's rubber band selection can toggle many elements.

3.2 Deriving selection domains from user events

Selection domains are determined based on points a user indicates with a pointing device, keyboard, or by tapping. Often only two points, the anchor and active end, are of interest but

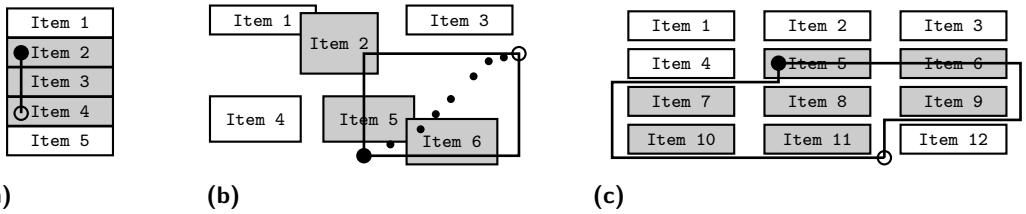


Figure 4 Illustration of concepts encountered in determining the selection domain. The solid circles indicates where the user has placed the anchor (with a click) and the hollow circles the active end (with a shift-click or a rubber band selection). Dots indicate other points on the selection path, detected during a mouse drag. In these examples only anchor and active end are used.

in general arbitrarily many points may be used. For example, in "lasso" selection, discussed briefly in Section 4.9, the user-indicated points are the vertices of a polygon and the selection domain consists of the elements that intersect with that polygon.

We call the sequence of points that determine a selection domain the *selection path*, and define the *anchor* as the first and the *active end* the last element of this path. In a one-element selection path, the anchor and active end coincide.

How the selection path maps to a set of elements depends on the application. Figure 4 shows three examples. In Figure 4a, the anchor and active end indicate the endpoints of a range of elements; in 4b, the opposite corners of a rectangle with which the selected elements intersect; and in 4c, a range in a row-wise ordering. The last behavior appears, e.g., in text editors, photo organizers, and File Explorer's two-dimensional views.

The anchor and active end, and in general the points on the selection path, are not necessarily mouse coordinates but rather points in some space V that we call the *selection space*. This space can be any coordinate system to which the visual locations of the elements and the user-indicated points can be translated to. For example, in Figure 4a, V is a totally ordered set of element indices, and a mouse coordinate maps to the index of the element that contains it; in 4b, V is the window's coordinate system; and in 4c, the points in V could be tuples that store both a mouse coordinate and the element index, as some points lie outside of any element. We name the function that performs the conversion from mouse coordinates to selection space coordinates $\mathsf{m2v}$; it is provided by the application programmer.

We can now capture all the variation in how different applications allow the user to indicate elements into one function $\mathsf{sdom} : V^* \to \mathcal{P}(I)$ that computes a selection domain from the selection path. Similarly to $\mathsf{m2v}$, sdom is provided by the application programmer. We use the term *selection geometry* for the set of functions that define the application or context-specific aspects of multi-selection.

In Figure 4a, sdom computes the range of indices between the anchor and active end; in 4b, sdom maps a rectangle to the indices of other rectangles it intersects with; and in 4c, sdom finds two elements, one closest (by some suitable definition) to the anchor and the other to the active end, and returns the range of indices between those elements.

Viewing collections of elements as indexed families lets us "abstract away" the elements. The sdom function in turn abstracts away the indices: since the selection domain J is determined by some sequence of points P in the selection space, the primitive selection operations op_J^f can be defined in terms of P, as $\mathsf{op}^f_{\mathsf{sdom}(P)}$. A sequence $\langle f_1, P_1\rangle, \ldots, \langle f_n, P_n\rangle$ of selection function–selection path pairs thus determines the composition of primitive selection operations $\mathsf{op}^{f_n}_{\mathsf{sdom}(P_n)} \circ \ldots \circ \mathsf{op}^{f_1}_{\mathsf{sdom}(P_1)}$ that can compute a new selection mapping.

$$
\begin{aligned}
\mathsf{click}_p &: \langle s, op, \_, \_\rangle && \mapsto \langle s, \mathsf{op}^{\lambda x.\,\mathsf{T}}_{\mathsf{sdom}(\cdot|p)} \circ \mathsf{op}^{\lambda x.\,\mathsf{F}}_{\{i \in I \mid op(s)(i) = \mathsf{T}\}} \circ op, \cdot|p, \mathsf{ae}(\cdot|p)\rangle \\
\mathsf{c\text{-}click}_p &: \langle s, op, \_, \_\rangle && \mapsto \langle s, \mathsf{op}^{\lambda x.\neg \mathsf{onsel}(p, op(s))}_{\mathsf{sdom}(\cdot|p)} \circ \mathsf{op}^{\lambda x.x}_{\varnothing} \circ op, \cdot|p, \mathsf{ae}(\cdot|p)\rangle \\
\mathsf{s\text{-}click}_p &: \langle s, \mathsf{op}^{f}_{\_} \circ op, P, \_\rangle && \mapsto \langle s, \mathsf{op}^{f}_{\mathsf{sdom}(P|p)} \circ op, P|p, \mathsf{ae}(P|p)\rangle \\
\mathsf{s\text{-}click}_p &: \langle s, op, \_, \_\rangle && \mapsto \langle s, \mathsf{op}^{\lambda x.\,\mathsf{T}}_{\mathsf{sdom}(\cdot|p)} \circ \mathsf{op}^{\lambda x.x}_{\varnothing} \circ op, \cdot|p, \mathsf{ae}(\cdot|p)\rangle
\end{aligned}
$$

(a) The three basic selection commands.

$$
\begin{aligned}
\mathsf{undo} &: \langle s, o_n \circ o_{n-1} \circ op, \_, p\rangle && \mapsto \langle s, op, \bot, p\rangle \\
\mathsf{undo} &: t && \mapsto t \\
\mathsf{bake} &: \langle s, op \circ o_4 \circ o_3 \circ o_2 \circ o_1, R, p\rangle && \mapsto \langle \mathsf{store}(s, o_2 \circ o_1), op \circ o_4 \circ o_3, R, p\rangle \\
\mathsf{bake} &: t && \mapsto t
\end{aligned}
$$

(b) Undo and bake operations.

$$
\begin{aligned}
\mathsf{space}' &: \langle s, op, P, p\rangle \mapsto \mathsf{click}_p(\langle s, op, P, p\rangle) && \mathsf{arrow}'_{dir} : \langle s, op, P, p\rangle \mapsto \langle s, op, P, \mathsf{step}_{dir}(p)\rangle \\
\mathsf{s\text{-}space}' &: \langle s, op, P, p\rangle \mapsto \mathsf{s\text{-}click}_p(\langle s, op, P, p\rangle) && \mathsf{s\text{-}arrow}'_{dir} = \mathsf{s\text{-}space}' \circ \mathsf{arrow}'_{dir} \\
\mathsf{c\text{-}space}' &: \langle s, op, P, p\rangle \mapsto \mathsf{c\text{-}click}_p(\langle s, op, P, p\rangle) && \mathsf{c\text{-}arrow}'_{dir} = \mathsf{arrow}'_{dir} \circ \mathsf{c\text{-}space}'
\end{aligned}
$$

(c) Keyboard selection operations.

Figure 5 The basic commands of the selection language.

4 Selection language

We give precise meanings to multi-selection features by defining the components that constitute the state of a selection and the functions that define the valid transformations on that state. These definitions are a language for implementing multi-selection features. The language is parametrized by a selection geometry, a handful of functions that the programmer defines. As a heads up, in addition to the already introduced sdom and m2v, the other functions of the selection geometry are step, default, and $|$, and a function to filter elements of the index set I based on a predicate.

Figure 5 defines the basic commands of the selection language. Each command is a function that maps the current selection state to a new state. The function definitions rely on pattern matching to express concisely the complete effect that a command has on the selection state. If more than one pattern matches, the first matching definition applies.

The selection state is a tuple of the form $\langle s, op, P, p\rangle$, where $s : I \to \mathbf{2}$ is the base selection mapping; op the composition of primitive selection operations that have not yet been applied and stored to s; P the selection path; and p the *keyboard cursor*, explained in Section 4.4.

The third element of the tuple has two uses. In lieu of a selection path, it can contain a *selection predicate*. The metavariable P always signifies a path and the metavariable Q a selection predicate; selection predicates are explained in Section 4.5. The third element can also have an undefined value $\bot$. Neither metavariable P nor Q matches $\bot$. The metavariable R matches any of these three kinds of values.

The selection mapping that reflects the currently selected elements is obtained from op and s as $op(s)$: the selected elements are $\{i \in I \mid op(s)(i) = \mathsf{T}\}$. We consider the empty composition, denoted as $op = \cdot$, to be the identity function, so that $\cdot(s) = s$. The metavariable o ranges over primitive selection operations and op over compositions of zero or more primitive selection operations. While function composition has its regular meaning, we also assume that the composition structure is accessible, so that pattern matching can

extract individual primitive selection operations. For example, the pattern $o_1 \circ o_2$ matches compositions that have exactly two primitive selection operations, binding o_1 to the first and o_2 to the second. The metavariable *op* can appear on either end of a pattern: $op \circ o$ and $o \circ op$ both match compositions that have one or more primitive selection operations. Finally, the metavariable _ binds to anything and signifies an unused value.

A selection path is a sequence of points in V. The symbol $\cdot$ denotes the empty sequence. The operator | is one of the context-dependent functions whose meaning can vary, but to simplify, we assume for now that it extends a sequence with a new point. For example, if $P = p_1, \ldots, p_n$, then $P|p$ is the sequence $p_1, \ldots, p_n, p$. The function $\mathsf{ae}(P)$ returns the active end, the last element of a sequence, or the special undefined value $\perp$ if P is the empty sequence. We thus overload $\perp$ to signify either an undefined point or an undefined path/selection predicate. We assume that we can query whether a value is undefined by matching against the pattern $\perp$.

All functions of the selection language maintain the invariant that the composition of primitive selection operations has an even number of elements. This is a technicality that makes the definition of the language more concise at the expense of sometimes adding extra empty operations to the composition. A possible valid initial empty selection state that satisfies the invariant is $\langle e, \cdot, \perp, \perp \rangle$, where e is the empty selection mapping.

4.1 Meaning of mouse clicks

Figure 5a shows our definitions for the three basic operations for constructing selections: click, shift-click, and command-click. The point parameter p to each operation is written as a subscript, as in click_p, to keep it notationally separate from the selection state parameter. The click functions do not modify the base selection mapping s; the current selection mapping $op(s)$ changes because the composition *op* is modified.

The click_p function adds two primitive selection operations to *op*. The first operation clears the current selection; its selection domain is the currently selected elements and not I, which might be difficult to express, even unbounded in some selection geometries. The second operation selects the element(s) indicated by p, if any. The previous selection path is discarded and p becomes the only element of the new selection path, and thus both the anchor and active end. It also becomes the keyboard cursor. If elements can overlap in V, the definition of sdom may need to handle this one-element selection path specially. The established behavior is that a click selects at most one element. Hence, $\mathsf{sdom}(\cdot|p)$ should either return the singleton set whose element is the index of the "topmost" element that p could indicate, or the empty set if p indicates no elements.

The $\mathsf{c\text{-}click}_p$ function is similar to click in that it clears the selection path and sets a new anchor. If p indicates an element, command-click should toggle that element. Instead of using the toggling selection function $\lambda x.\neg x$, $\mathsf{c\text{-}click}_p$ uses either $\lambda x.\,\mathsf{F}$ or $\lambda x.\,\mathsf{T}$, depending on whether p is on a selected or unselected element. This is to establish the mode of selecting, i.e., whether subsequent shift-clicks should select or deselect. The helper function $\mathsf{onsel}(p, s) = \textbf{if}\ \mathsf{sdom}(\cdot|p) = \{i\}\ \textbf{then}\ s(i)\ \textbf{else}\ \mathsf{F}$ determines whether p is on a selected element. One primitive selection operation would suffice to capture the effect of a command-click, but to maintain the invariant of an even number of primitive selection operations, command-click first adds the identity operation $\mathsf{op}^{\lambda x.x}_{\varnothing}$ to the composition.

Shift-click is defined by two cases. The first matches if both the *op* composition is not empty and the selection path is defined. The second matches all tuple values; it is thus selected if *op* is empty or if the tuple's third component is not a path (if it is $\perp$ or a selection predicate). In the first case, the point p is added to the selection path as the new active

end. A new selection domain is computed from this path to replace the selection domain of the outermost primitive selection operation in the composition. Shift-click thus cancels the current active domain and establishes a new one. There may be no selection path to extend or no outermost primitive selection operation, so in the second case a new path is established and a new pair of primitive selection operations added. The effect is the same as that of a command-click, except that the mode of selecting is always set to select.

All three click functions extend the path with p, then use ae to access the path's active end and make it the new keyboard cursor. This may seem like a roundabout way of setting the cursor to p, and indeed it usually is. However, since the | operator is a parameter to the selection language, it might be defined to sometimes not make p the active end. For example, in some selection contexts it is useful to ignore certain points, such as coordinates outside of any selectable elements. This is easily modeled by defining | to keep the path unchanged when called with such a point. To enable this customization point, all selection path manipulation is performed with the | operation. The reason for setting the cursor using ae is now plain: when a point is ignored, we get the desired effect that click and command-click set the keyboard cursor to $\perp$ and shift-click keeps it unchanged.

The | function can actually be defined to modify the selection path in arbitrary ways. For example, when only the anchor and active end are of interest, | can limit the length of the selection path to two elements, throwing away the intermediate points. As another example, one of our example applications defines a selection geometry where | may remove points from the end of the selection path to get the effect of erasing points of a lasso selector.

We consider the above three operations to be a sufficient set for convenient manipulation of the selection mapping with a pointing device. Compared to File Explorer and Finder, an operation for toggling many elements simultaneously is missing. It would not be problematic to specify or implement but, as discussed in Section 2.5, its use cases are not obvious.

4.2 Meaning of rubber band selection

In existing practice, rubber band selection and selecting via mouse clicks often have different semantics. For example, Finder's rubber band selection always toggles, instead of selecting or deselecting. We equate mouse drag to shift-click, following old apparently forgotten advice [2, 12]. More precisely, the meaning of "drag mouse to point p" is exactly that of s-click$_p$; every *mousemove* event until a *mouseup* should be interpreted as a shift-click. The result of this equivalence is the interchangeability of the two selection mechanisms.

To appreciate the benefits, consider a rubber band selection that selects a region that extends beyond the current window's visible area. Many user interfaces scroll the window when the mouse is dragged past the window's edge. It is easy to "overshoot", then be forced to scramble back by dragging the mouse to the opposite edge of the window, possibly overshooting again, and at all times being careful not to release the mouse button. When shift-click has the same meaning as mouse drag, one can let go of the mouse at any point, scroll by whatever method is convenient (scrollbar, page down key, two-finger scroll, etc.), and use shift-click to finish the selection or pick up the rubber band and continue the drag. MS Word supports this behavior in text selection and Excel in selecting spreadsheet cells.

4.3 Undo, redo, and baking

In the presented framework, undoing selection commands means simply removing outermost primitive selection operations from the composition. Figure 5b shows the undo function.

Every undoable selection command adds two operations to the composition, so undo removes two operations at a time.

The undo function sets the selection path to the undefined value $\perp$. This is because undoing exposes a new primitive selection operation whose selection domain was not computed from the current selection path (or predicate), and thus keeping the current path would make a subsequent shift-click command very unpredictable. An undefined path forces shift-click to add a new pair of primitive selection operations, avoiding the surprising behavior. Note that undo preserves the keyboard cursor; there are no surprises in doing so. The second case of undo is an identity function; it is applied if there are no operations to undo.

To support redo, the operations removed from the composition by undo can be stored to a stack of redoable selection operations, to be composed back when requested. We do not show the definitions. Our reference implementation, described in Section 5, implements redo.

There are no established key bindings for undo/redo of selections, since such a feature is seldom, if ever, provided. Adopting typical undo/redo bindings, such as command-Z/shift-command-Z, would mean coalescing the selection undo stack and the application's command undo stack, making selection undo subordinate to command undo: executing a command on the selection (such as deleting or copying the selected files) would clear the selection undo stack. Selections would be undoable only until the most recent command. We have not explored all the ramifications, and thus keep selection undo separate from command undo. We bind option-Z and shift-option-Z to selection undo and redo, respectively.

Every new click or command-click operation grows the composition of primitive selection operations. For reducing the size of the composition, without changing which elements are selected, we introduce the bake function in Figure 5b. It extracts the two least recently added primitive selection operations from *op* and "bakes" their effect permanently to the base selection mapping s; we assume that $\mathsf{store}(op, s)$ is an operation that constructs this new base selection mapping from *op* and s. The second case of bake is an identity function; it is applied if fewer than two primitive selection operations would remain in *op* after baking. Emptying the composition completely would bake the operation that defines the active domain and thus change, e.g., shift-click's behavior. To impose a limit on the size of the composition, an implementation calls bake whenever a predefined upper limit is reached.

4.4 Meaning of keyboard selection operations

When navigating with arrow keys is meaningful, constructing selections with the keyboard should be possible and convenient. As discussed in Section 2.4, applications' support for keyboard selection varies considerably. This is expected since selection features are not the only candidates competing for the relatively small set of convenient key combinations. We choose a particular set of key bindings and give a few arguments to justify them below, but recognize that they are not suitable everywhere. Again, the contribution is the precise meanings of the keyboard selection operations, not particular key bindings.

The keyboard selection functions, named to suggest our key bindings, are shown in Figures 5c and 6. The functions do not manipulate the selection path or call sdom directly; they are all defined in terms of the three click functions. The point argument they pass to the click functions is the current keyboard cursor, which is a point in the selection space V.

Assuming p is the keyboard cursor, space' has the effect of click_p, $\mathsf{s\text{-}space}'$ the effect of $\mathsf{s\text{-}click}_p$, and $\mathsf{c\text{-}space}'$ the effect of $\mathsf{c\text{-}click}_p$. Note the "primed" function names in Figure 5c. These functions assume that the keyboard cursor is defined. If, however, no click operations have yet been applied, the cursor may have its initial undefined value. Figure 6 shows the corresponding functions that accept $\perp$ as the keyboard cursor; these are the functions to

$$\mathsf{space}' : \langle s, op, P, p\rangle \mapsto \textbf{if } \mathsf{crs_s}(p) \neq \bot \textbf{ then } \mathsf{space}(\langle s, op, P, \mathsf{crs_s}(p)\rangle) \textbf{ else } \langle s, op, P, p\rangle$$

$$\mathsf{s\text{-}space}' : \langle s, op, P, p\rangle \mapsto \textbf{if } \mathsf{crs_s}(p) \neq \bot \textbf{ then } \mathsf{s\text{-}space}(\langle s, op, P, \mathsf{crs_s}(p)\rangle) \textbf{ else } \langle s, op, P, p\rangle$$

$$\mathsf{c\text{-}space}' : \langle s, op, P, p\rangle \mapsto \textbf{if } \mathsf{crs_s}(p) \neq \bot \textbf{ then } \mathsf{c\text{-}space}(\langle s, op, P, \mathsf{crs_s}(p)\rangle) \textbf{ else } \langle s, op, P, p\rangle$$

$$\mathsf{arrow}_{dir} : \langle s, op, P, p\rangle \mapsto \textbf{if } p \neq \bot \textbf{ then } \mathsf{arrow}'_{dir}(\langle s, op, P, p\rangle) \textbf{ else} \langle s, op, P, \mathsf{crs}_{dir}(p)\rangle$$

$$\mathsf{s\text{-}arrow}_{dir} : \langle s, op, P, p\rangle \mapsto \begin{cases} \mathsf{s\text{-}arrow}'_{dir}(\langle s, op, P, p\rangle) & \text{if } p \neq \bot \\ \langle s, op, P, \bot\rangle & \text{if } \mathsf{crs}_{dir}(p) = \bot \\ \mathsf{s\text{-}space}'(\langle s, op, P, \mathsf{crs}_{dir}(p)\rangle) & \text{otherwise} \end{cases}$$

$$\mathsf{c\text{-}arrow}_{dir} : \langle s, op, P, p\rangle \mapsto \begin{cases} \mathsf{c\text{-}arrow}'_{dir}(\langle s, op, P, p\rangle) & \text{if } p \neq \bot \\ \langle s, op, P, \bot\rangle & \text{if } \mathsf{crs}_{dir}(p) = \bot \\ \mathsf{c\text{-}space}'(\langle s, op, P, \mathsf{crs}_{dir}(p)\rangle) & \text{otherwise} \end{cases}$$

$$\mathsf{crs}_{dir} : p \mapsto \textbf{if } p = \bot \textbf{ then } \mathsf{default}_{dir} \textbf{ else } p$$

Figure 6 Arrow and space commands with default keyboard cursor positions.

call from client code. We discuss the primed functions first, as they explain the meaning of keyboard commands in relation to the basic click functions without confusion.

We adopt spacebar as the keyboard equivalent of clicking, following the example of File Explorer (which does not, however, bind the unmodified spacebar to a selection command). Many applications bind space to other events: Finder applies the "open" command to the selected file(s) and Gmail delegates to the default binding in browsers to scroll down the screen. It is not critical if spacebar is too contested to be used for selection. The arrow keys with the bindings we propose are sufficient for selecting multiple disjoint ranges of elements.

The arrow functions are parameterized by a direction, where $dir \in \{\mathsf{left}, \mathsf{right}, \mathsf{up}, \mathsf{down}\}$, so that one definition covers all four arrow keys. The arrow'_{dir} functions are for arrow keys without modifiers. They do not change the selection state of any element. They move the keyboard cursor to the specified direction according to the step function discussed below. We find it natural that arrow keys move the cursor freely. This is the convention in text editors and it is analogous to the mouse moving the mouse pointer. This choice, however, does not optimize for selecting a single element, which is arguably the most common need.

Arrows with modifiers predictably combine a keyboard cursor move with a similarly modified selection operation: s-arrow′ function is the composition of the s-space′ and arrow′ functions, and c-arrow′ the composition of arrow′ and c-space′ functions. Importantly, with the command modifier the selection operation is applied first and move second; with the shift modifier move is applied first. The intuition is that with the command modifier one indicates where a selection region starts and with shift where it ends.

4.4.1 Keyboard selection and coordinates

The arrow function uses the function $\mathsf{step} : \langle\{\mathsf{left}, \mathsf{right}, \mathsf{up}, \mathsf{down}\}, V\rangle \to V$ to determine how the cursor moves in response to arrow keys. This function depends on the coordinate system of the selection space V and on the placement of elements in V. It is one of the functions of the selection geometry and must be defined by the application programmer.

In a grid layout, the definitions of all directions are obvious. In a row-wise sequential ordering (text editors, for example), left and right are trivial, up moves to a position on the line above the current cursor position and down on the line below. Rules for which position on a line to move to depend on the application. E.g., text editors typically choose the

position with the smallest difference along the x-axis, except when the cursor is on the last position of a line, in which case the last position of the new line is chosen as the new cursor location. When elements can be positioned in arbitrary locations, the step function can be rather complex. It may, for example, examine a sector from the cursor towards the selected direction for candidate elements and choose one according to some criteria. In Finder, an element's fitness seems to increase when the angle from the requested direction and distance from the current element decrease.

4.4.2 Keyboard cursor defaults

A selection context usually starts with no keyboard cursor set, and thus a mechanism for defining default values for the cursor is necessary. The most natural default may vary depending on the issued keyboard command. For example, in a vertical list of elements, arrow down might set the cursor to the first element, arrow up to the last element, and spacebar might not have a default at all. We rely on the selection geometry's function $\mathsf{default}_{dir}$ to provide the default keyboard cursor value for each of the four arrow keys, and additionally for the direction s, which we use to mean the spacebar. Thus here $dir \in \{\mathsf{left}, \mathsf{right}, \mathsf{up}, \mathsf{down}, \mathsf{s}\}$. If some direction d has no default, then $\mathsf{default}_d$ should return $\perp$.

Figure 6 defines the keyboard selection functions to be used by the client. They inspect the cursor, and, if it is undefined, try to establish it by invoking the default function. If the cursor remains undefined, the keyboard functions have no effect. The space functions are simple wrappers over the corresponding primed functions in Figure 5c. The arrow functions are a bit more complex; they only delegate to the corresponding primed function if the keyboard cursor is defined. If the cursor is initialized with a default value, the corresponding primed function is not invoked; it could be confusing if the cursor was first set to the default, then immediately moved by the primed function. This is why s-arrow and c-arrow, after establishing a cursor from a default value, forward to s-space′ and c-space′, respectively.

Supporting default cursor values adds some complexity to the keyboard functions, but the additional definitions are mostly repetition of the same wrapping pattern. Importantly, it is very easy for the application programmer to add default cursor positions to a multi-selection feature: the default function is essentially a lookup table of at most five key-value pairs.

4.5 Selecting based on a predicate

Apart from selecting elements with a pointing device, a user may wish to select or deselect elements based on their properties. For example, a user may want to select all files whose name matches a particular string. To integrate a predicate-based selecting or deselecting into the selection language, the *selection predicate* takes a similar role as the selection path, and determines a selection domain. Selecting or deselecting is then accomplished in the usual manner of adding primitive selection operations to the composition.

Figure 7 defines two operations, $\mathsf{predicate\text{-}select}^b_Q$ and commit, where $Q : I \to \mathbf{2}$ is a predicate on the indices, and $b \in \mathbf{2}$ determines whether the operation selects or deselects. The selection domain that Q defines is $\{i \in I | Q(i)\}$. The selection language does not dictate the representation of I, so we assume that the selection geometry implements an efficient way to iterate over I and find the elements that satisfy Q.

The predicate-select command works analogously to shift-click: if a predicate has already been defined (the previous command was predicate-select), the topmost primitive selection operation is replaced with one whose selection domain is computed using the new predicate Q. The first rule matches when a predicate is defined because the metavariable Q' only

$$
\begin{array}{rl}
\mathsf{predicate\text{-}select}^{b}_{Q}\colon & \langle s, \_ \circ op, Q', p\rangle \mapsto \langle s, \mathsf{op}^{\lambda x.b}_{\{i\in I|Q(i)\}} \circ op, p\rangle \\
\mathsf{predicate\text{-}select}^{b}_{Q}\colon & \langle s, op, \_, p\rangle \mapsto \langle s, \mathsf{op}^{\lambda x.b}_{\{i\in I|Q(i)\}} \circ \mathsf{op}^{\lambda x.x}_{\varnothing} \circ op, p\rangle \\
\mathsf{commit}\colon & \langle s, op, \_, p\rangle \mapsto \langle s, op, \bot, p\rangle
\end{array}
$$

Figure 7 The commands for selecting with a predicate.

matches predicates, not paths or $\bot$. If there is no predicate (the third tuple component is either $\bot$ or a selection path), the second rule applies and a new pair of primitive selection operations is added.

The commit function makes the effect of a predicate-selection permanent, in the sense that a new undoable state is created. This is accomplished by setting the current predicate to $\bot$, so that a subsequent predicate-selection command will add a new pair of primitive selection operations before applying a predicate. Note that even though shift-clicking behaves quite similarly to predicate-selection, only the latter needs a dedicated commit command; an active domain resulting from shift-clicks can be made permanent with a command-click.

Using the same component of the selection state tuple for both selection paths and predicates may seem confusing. It rather naturally guarantees, however, that shift-clicks and predicate-selection commands do not interfere with each other. Since predicate-selection considers anything other than a predicate the same as undefined, and shift-click anything other than a selection path the same as undefined, both a predicate-selection command after a shift-click and a shift-click command after predicate-selection will always add a new pair of primitive selection operations, as they should.

4.6 Summarizing the selection language

The definitions above are the complete selection language. They define the meaning of selection operations to be bound to the click, shift-click, and command-click mouse events, as well as to the space, shift-space, command-space, arrow, shift-arrow, and command-arrow keyboard events. They define the undo and redo operations and selection based on elements' properties. The language is parametrized by the functions sdom, m2v, step, default, and $|$, and a filtering function. Implementing these six functions provides a selection model with all the features expected of a full-fledged selection feature, and probably more (e.g., undo).

Section 5 describes a concrete implementation of the selection language, but below we first touch on a few additional aspects of it.

4.7 Touch gestures

As discussed in Section 2.6, multi-selection in touch platforms is a wild-west, with a wide variety of gestures and meanings for those gestures. Adapting our model to touch platforms is mostly future work, but we see no reasons not to provide the same basic selection operations with the same semantics in both touch and non-touch platforms. Due to no good way of modifying taps (as with shift or command), the challenge is to identify natural gestures that can be given the meanings of click, shift-click, and command-click operations of our selection language, and that can continue as a rubber band selection.

4.8 Active domain

Section 2.5 pointed out two different responses to the shrinking of the active domain. The question is what should be the selection state of the elements that were in the active domain before the active end changed but are not in it anymore. Finder's answer is "unselected" if the active end was changed as the result of a shift-click, and "whatever the state was before" if it was changed via a mouse drag. This difference seems arbitrary. Our view is that regardless of the means of selection, the prior selection state should always be restored. Otherwise it is easy to unintentionally destroy existing selection state by shift-clicking or dragging the mouse "too far". Furthermore, the use case for not remembering the prior selection state under the active domain is contrived. It can be distilled to the user specifying some active domain J_1, then shift-clicking to change it to J_2, and expecting all elements in J_2 to become selected and those in $J_1 \setminus J_2$ deselected—this is not an intuitive operation.

It is easy to see that our selection operations remember and restore elements' prior selection state. Let s be some base selection mapping, and $\mathsf{op}^f_{J_1} \circ op$ the current composition of selection operations. The current selection mapping is then $(\mathsf{op}^f_{J_1} \circ op)(s)$ and the active domain is J_1. If the user changes the active end to J_2, either by shift-click, mouse drag, shift-arrow, shift-space, or predicate-selection, then the new selection mapping is $(\mathsf{op}^f_{J_2} \circ op)(s)$; any effect that $\mathsf{op}^f_{J_1}$ had on the selection state is completely canceled.

4.9 Selections that use the entire selection path

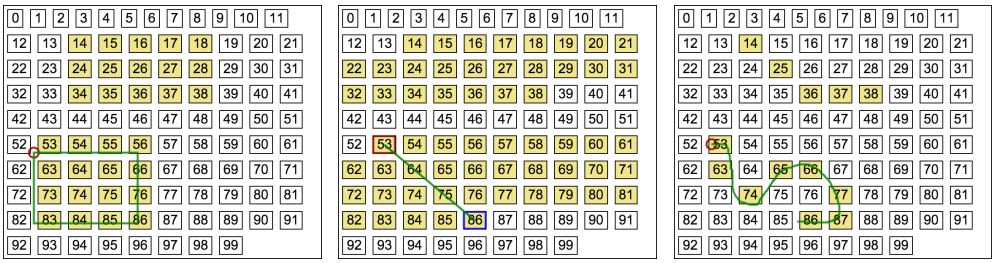
All our examples thus far have assumed that the selection domain is determined by at most two selection space points: the anchor and active end. Yet the functions in Figure 5 maintain an arbitrarily long selection path, with each shift-click operation adding more points to it We wrote the formalism, and our implementation, in this manner to be able to support selection geometries where more than two points are needed to define the selection domain. For example, in "lasso" selection the points of an entire mouse drag are taken as the vertices of a polygon; items that intersect with the polygon belong to the selection domain. Since the entire selection path is passed as input to the sdom function, supporting lasso selection is simply a matter of defining sdom appropriately.

5 Packaging selection behavior into a library

We provide a reference implementation of the selection language as a JavaScript library at `http://hotdrink.github.io/multiselectjs`. It is a faithful implementation of the specification, though on the surface there are a few differences; we point them out below. The library comes with a tutorial and example applications. The examples show concretely the division of labor: the tasks required of the application programmer vs. the services that the library provides. Further, they make it clear that the selection semantics remains the same in all selection contexts, even if the selection geometry is different. The library allows switching between selection geometries on-the-fly without losing the current selection state. Figure 8 shows three snapshots from an example application packaged with our library; each snapshot is taken when a different selection geometry object was in use.

5.1 SelectionState class

The central abstraction in the library is the SelectionState class (we emulate classes by objects with the usual JavaScript conventions). The elements of the selection state tuple are member



(a) **(b)** **(c)**

Figure 8 Snapshots of selecting using three different selection geometries: (a) rectangular, (b) row-wise, and (c) "snake" (all elements that touch the selection path are included in the selection domain). User's mouse events were the same in each case: click at element 14, shift-click at 36, shift-click at 38, command-click near or at 53, and mouse drag to near or at 86. The rubber band indicator is visible because the snapshots were taken prior to releasing the mouse to end the drag.

variables of this class. Every command of the selection language is a method of SelectionState. Unlike the functions of the selection language that each map a selection state tuple to a new selection state tuple, SelectionState class's methods modify the encapsulated state, as is conventional in object-oriented libraries. The implementation includes a few optimizations:

- The current selection state of an element i is $op(s)(i)$, where op is the current composition and s the base selection mapping. If op contains many primitive selection operations, evaluating i's selection state would require many function calls. For example, if the composition object represents the value $\mathsf{op}^f_{J_1} \circ \mathsf{op}^g_{J_2} \circ \mathsf{op}^h_{J_3}$, the straightforward way to compute $(\mathsf{op}^f_{J_1} \circ \mathsf{op}^g_{J_2} \circ \mathsf{op}^h_{J_3})(s)(i)$ is $\mathsf{op}^f_{J_1}(\mathsf{op}^g_{J_2}(\mathsf{op}^h_{J_3}(s)))(i)$. If, however, f is a constant function ($\lambda x.\,\mathsf{T}$ or $\lambda x.\,\mathsf{F}$) and $i \in J_1$, then it suffices to compute $\mathsf{op}^f_{J_1}(i)$. As another example, if $i \notin J_1$ and $i \notin J_2$, then it suffices to compute $\mathsf{op}^h_{J_3}(i)$. The implementation of the composition maintains additional information about which selection functions are constant and which selection domains include a particular index, so that unnecessary evaluations like the ones above are avoided.
- The semantics of shift-clicking guarantees that the effect of two consecutive shift-clicks at points p_1 and p_2 is the same as first extending the selection path with p_1, then shift-clicking at p_2. The library takes advantage of this property when many shift-click events happen in rapid succession, e.g., during a rubber band selection, and coalesces several consecutive shift-click calls into one so that the selection domain is not computed after every mouse location visited during a drag.

The size of the selection state object is proportional to the total number of elements in all of the primitive selection operations' domains. The number of primitive selection operations is limited by the number of allowed undo states set by the programmer.

We chose to maintain the composition of primitive selection operations as functions, which is faithful to the presented formalism. Another feasible alternative is to store, as sets of indices, the "diffs" between the applications of each two consecutive primitive selection operations. To represent selection domains and the baked selection mapping, we use JavaScript's Map container, where each element's selection status is a separate entry. In some applications (selecting pixels in images, for example), other representations, such as bitvectors or run-length encoded sets, may lead to a smaller memory footprint and faster manipulation of selections. The library is parameterized on the above two aspects, that is, on how the

composition and selection domains are represented. These parameterizations are purely for performance reasons and do not affect semantics. They are not discussed in this paper.

5.2 Selection Geometry Classes

Selection state objects are parameterized by a selection geometry object that captures all context-specific aspects of multi-selection. A selection geometry object thus has a corresponding method for each of the functions sdom, m2v, step, default, and |, introduced in Section 4, as well as for selecting elements based on a predicate:

- selectionDomain(path) to compute a set of indices from the selection path;
- m2v(point) to translate mouse coordinates to selection space;
- step(direction, point) to determine the cursor movement after an arrow key press;
- defaultCursor(direction) to provide the default cursor value for each direction;
- extendPath(path, point) to append point to path; and
- filter(predicate) to find the set of indices that satisfy predicate.

Figure 9 shows a complete implementation of a selection geometry for the case of rectangular elements that can be positioned arbitrarily. The selectionDomain function iterates over all elements and returns those that intersect with the rectangle formed by the anchor and active end. If the selection path has exactly one element, at most one element is returned. This is so that a click or command-click only selects the topmost element when the clicked point is on two or more overlapping elements. We assume the topmost element is the one with the smallest index. The m2v function is the identity function, since the selection space and mouse coordinates coincide. The extendPath function discards the selection path's intermediate points, since only the first and last points (the anchor and active end) are used in computing the selection domain. The step and defaultCursor are not be needed since keyboard commands are not supported; we include stubs for them for completeness. In practice, geometry classes would inherit from ms.DefaultGeometry, which provides default definitions that apply for many geometries. For RectangularGeometry it would suffice to implement just the selectionDomain method, and filter if predicate-selection needed to be supported.

One might want to optimize selectionDomain so that it does not iterate over all elements on every call. E.g., the selectable area could be divided into segments and the iteration limited to the elements in segments that intersect with the selection path. Also, the library makes the previous selection domain object available to selectionDomain, so that selection domain computations can be incremental. We do not discuss these aspects further in this paper.

Figure 10 shows another selection geometry class. A geometry like this could be used to implement multi-selection in a list box, Finder's list view, list of emails on a mail client, etc. The selection space coordinates are now indices of the elements in the elements array. The selectionDomain function thus returns the elements within the range between the anchor and the active end. The m2v function finds the index of the element that contains a given mouse coordinate. Clicks outside of all elements might be possible; we choose to return the selection space coordinate **null** for such mouse coordinates, and define extendPath to ignore **null** coordinates. Otherwise extendPath's implementation is the same as in RectangularGeometry above. To implement keyboard navigation, the step method recognizes the ms.UP and ms.DOWN directions, and decrements and increments the keyboard cursor respectively. The defaultCursor implements cursor defaults so that up-arrow starts from the bottom and down-arrow from the top. Filtering is as in RectangularGeometry.

```
function RectangularGeometry(elements, bbox) {
  this.selectionDomain = function(path) {
    var J = new Map();
    var r1 = makeRectangle(ms.anchor(path), ms.activeEnd(path));
    for (var i=0; i<elements.length; ++i) {
      var r2 = bbox(elements[i]);
      if (rectangleIntersect(r1, r2)) { J.set(i, true); if (path.length == 1) return J; }
    }
    return J;
  };
  this.m2v = function(p) { return p; };
  this.extendPath = function (spath, p) { if (spath.length == 2) spath[1] = p; else spath.push(p); };
  this.step = function (dir, p) { return undefined; };
  this.defaultCursor = function(dir) { return undefined; };
  this.filter = function (pred) {
    var J = new Map();
    for (var i=0; I<elements.length; ++i) { if pred(i) J.set(i, true); }
    return J;
  };
};
```

Figure 9 An implementation of a selection geometry for selecting arbitrarily positioned rectangular elements. The constructor's elements parameter is the indexed family of elements, implemented as an array. The bbox function is assumed to compute a bounding box given an element. The helper functions makeRectangle and rectangleIntersect are what their names suggest. The ms namespace prefix indicates functions and constants in our library's API. Their names explain their purpose.

5.3 Visualizing selections

How selections are visualized is application dependent, when these visualizations take place is controlled by the library. Concretely, SelectionState's methods invoke a callback when the elements' selection states may have changed. Assuming CSS code defines suitable rendering, the callback can simply toggle elements' class attribute based on their selection state:

```
function refresh(sel) {
  var s = sel.selected();
  for (var i=0; i<elements.length; ++i) elements[i].className = s.get(i) ? "selected" : "unselected";
};
```

The parameter to the callback is the selection state object itself, whose selected method gives the currently selected indices. A selection state object can be requested to track changes to the elements' selection state. The refresh function will then receive the set of changed indices as a second argument.

5.4 Constructing a selection state object and registering event handlers

Assuming elements is an array of DOM objects, we can construct the selection geometry, then use that and the refresh function to construct a selection state object:

```
var geometry = new RectangularGeometry(elements, function(e) { return e.getBoundingClientRect(); });
var selection = new SelectionState(geometry, refresh);
```

To put the selection object to use, what remains to do is to recognize mouse and keyboard events and handle them by invoking selection's methods. The same mouse operations are often harnessed for many different tasks, which makes their event handling code complex and difficult to reuse. For example, a mouse click may invoke a command (such as opening

```
function VerticalListGeometry(elements, bbox) {
  this.selectionDomain(path) {
    var J = new Map(), a = ms.anchor(path), b = ms.activeEnd(path);
    for (var i=Math.min(a, b); i<=Math.max(a, b); ++i) J.set(i, true);
    return J;
  };
  this.m2v = function(p) {
    for (var i=0; i<elements.length; ++i) if (pointInRectangle(p, bbox(elements[i]))) return i;
    return null;
  };
  this.extendPath = function (spath, p) {
    if (p == null) return;
    if (spath.length == 2) spath[1] = p; else spath.push(p);
  };
  this.step = function (dir, p) {
    if (dir == ms.UP) return Math.max(p - 1, 0);
    if (dir == ms.DOWN) return Math.min(p + 1, elements.length-1);
    return p;
  };
  this.defaultCursor = function (dir) {
    if (dir == ms.UP) return elements.length - 1;
    if (dir == ms.DOWN) return 0;
  };
  this.filter = function (pred) {
    var J = new Map();
    for (var i=0; I<elements.length; ++i) { if pred(i) J.set(i, true); }
    return J;
  };
};
```

Figure 10 An implementation of a selection geometry for vertically stacked elements (a list box, for example). The conventions are the same as in Figure 9. Here, we rely on one additional helper function, pointInRectangle, and use the constants UP and DOWN from our library.

a file), start a drag-and-drop operation, or indicate a selection. Distinguishing between different commands can be a nuanced task: a *mousedown* event followed by a *mouseup* could mean selection if the mouse did not move (too much) in between, otherwise a start of a drag-and-drop. We thus expect that the event handling that triggers selection commands may require some application-specific code. The SelectionState class's onSelected(p) method that determines whether the selection space point p is on a selected element or not helps in writing this code. For example, the method makes it easy to distinguish between a *mousedown* event on a selected element and a *mousedown* event elsewhere. The former is usually a possible start of a drag-and-drop of the selected elements and the latter a start of a selection.

Once the mouse interaction has been interpreted to mean a selection command, the code that effects the command is trivial: a call to a method in the SelectionState class. Figure 11 shows possible implementations of the event handlers for the *mousedown*, *mousemove*, and *mouseup* events. For simplicity, we assume that drag-and-drop is not supported. Keyboard event handlers are equally straightforward. They merely have to recognize the set of supported key presses and map each of them to an appropriate method call. To give one example, the press of a command-modified down-arrow would call selection.cmdArrow(ms.DOWN).

The handlers may also call functions to draw or clear the anchor, cursor, and rubber band indicators as appropriate. As these aspects vary from one selection context to another, implementing these functions are the application programmer's responsibility—the library merely makes readily available the necessary information, the selection path and cursor. As

```
function mousedownHandler (evt) {
  var p = selection.geometry().m2v(getMousePos(evt));
  switch (ms.modifierKeys(evt)) {
    case ms.NONE: selection.click(p); break;
    case ms.SHIFT: selection.shiftClick(p); break;
    case ms.CMD: selection.cmdClick(p); break;
  }
  drawAnchor(selection.selectionPath()); drawCursor(selection.cursor());
  document.addEventListener('mousemove', mousemoveHandler);
  document.addEventListener('mouseup', mouseupHandler);
}
function mousemoveHandler (evt) {
  var p = selection.geometry().m2v(getMousePos(evt));
  selection.shiftClick(p);
  drawCursor(selection.cursor()); drawRubber(selection.selectionPath());
}
function mouseupHandler (evt) {
  document.removeEventListener('mousemove', mousemoveHandler);
  document.removeEventListener('mouseup', mouseupHandler);
  clearRubber();
}
```

Figure 11 An example implementation of mouse event handlers for multi-selection: selection is the selection state object, getMousePos extracts the mouse position from the event data, ms.modifierKeys inspects the event data and determines whether the shift or command modifiers were held down, and the draw* and clear* functions do what their names suggest. The current geometry is accessible as selection.geometry(). So that the handlers are not too aggressive, only the handler for *mousedown* is registered continuously, the others are activated at *mousedown* and deactivated at *mouseup*. We omit event handling boilerplate that stops event propagation and prevents default actions.

explained in Section 5.1, the selection domain may not be updated for each mouse move and shift-click event. By invoking the cursor/anchor/rubberband drawing functions directly in the handler functions, these visual indicators are, however, always updated instantaneously.

5.5 Summary

The recipe to implement multi-selection in a particular context is as follows: (1) select a data structure for the indexed family of elements; (2) implement a "refresh" function to visualize the selection state of the elements; (3) implement a selection geometry class; (4) implement functions to draw the anchor, cursor, and rubber band; and (5) register event handlers for mouse and keyboard events and map them to calls to a selection object's methods.

Many of the "implement" tasks can be expected to be "reuse" tasks: visualizing selected/unselected status is perhaps nothing more than a function call to change the style or color property of an element; some of the geometry's functions could be reused from another geometry and the code for drawing the anchor and cursor indicators from another selection context; and the same event registration code can likely be used repeatedly.

In the best case scenario, implementing multi-selection for a given context means selecting the geometry and visualization functions to use, and turning the feature on. But even if nothing but the SelectionState class can be reused, the selection library and its abstractions simplify the application programmer's work in a fundamental way. Each item above is a well-specified and confined task. Each task involves defining one or more functions that perform a (usually straightforward and side-effect free) computation from inputs to outputs, realize a visual effect, or dispatch events to pre-defined handler functions. Such pieces of

code are predictable to develop and testable. There is no need to think about how selections change, how they should be updated, undone or redone; where anchors or cursors are located, when they should be moved, cleared, or drawn; or how or if mouse clicks, mouse dragging, and keyboard selection should work together. The findings described in Section 2 are evidence that these less tangible aspects are where the trouble lies in today's selection implementations.

6 Related work

One of the early guidelines for uniform multi-selection was for the Apple Lisa's user interface [1]. The operations of shift-click and mouse drag for selecting contiguous regions were defined, though not in much detail. The document that perhaps set the baseline for modern multi-selection, on Macintosh at least, is Apple's *Macintosh Human Interface Guidelines* [2, Ch. 10]. The notion of an anchor, how shift-click extends the selection by setting the active end of a range, interplay between mouse clicks, dragging, and keyboard selection operations are explained. Command-click is just presented as a toggling operation, not as establishing a new anchor, and nothing is said about restoring the selection state when the current selection shrinks. Similar guidelines for Windows [12] established File Explorer's selection behavior, though using shift-control-click for extending the selected range without disturbing other selections was not mentioned; that operation was bound to shift-click.

During the past three decades, instead of further progress towards a consistent and uniform multi-selection feature, the baseline set in the 80's has gradually deteriorated.

Today, there is no shortage of implementations of multi-selection features. Practically every GUI framework includes widgets for lists, tables, or trees of selectable items. The implementations, however, are intrusive and tightly coupled with a particular view or widget class. Typically, the programmer must wrap elements as a collection of a particular type of objects and place those objects in a particular view object, in order to get the "stock" multi-selection behavior "for free". Unless this is done, and in many cases it is not feasible due to the limitations of the provided view classes, the programmer is largely on their own.

For example, Apple's NSTableView and NSCollectionView classes provide Finder-like[2] list and icon multi-selection behavior [3], and for collections that fit to these views, their use is convenient. Their selection behavior, however, exhibits the problems described in Section 2. The corresponding iOS classes, UITableView and UICollectionView, are less useful, as their stock multi-select functionality is so minimal; as discussed in Section 2.6, each selected element requires an individual tap. But even if the multi-selection behavior provided by these classes were perfected, the main problem remains: the selection functionality is only available to the views implemented as instances of these specific classes. For example, Adobe Revel provides a "slot machine" view of images, where images are arranged in rows, each row representing an event. Each row can be of a different length and each row scrolls independently of other rows. We do not think it is possible to model this view in any of the predefined view classes.[3]

The situation is essentially the same with other widely used GUI frameworks. JavaFX's ListView and TableView have a pre-packaged selection feature. To build your own, the library provides the MultipleSelectionModel class [13], but it is not much more than an array of booleans where value changes trigger observable events. There are methods to select an individual

[2] The behavior is not exactly the same in icon view, since Finder itself does not use NSCollectionView.

[3] More generally in Adobe's software, which uses several GUI frameworks and Adobe's own platform-independent widgets, the lack of a reusable multi-selection feature has lead to many re-implementations. Symptoms are visible: e.g., the sequence of commands in Figure 1a would lead to different selections in Photoshop's "Paths" and "Layers" panel, and they would be different from the one produced in Finder.

index or a range of indices, and to clear all selections. The TableSelectionModel class adds the ability to arrange and select indices in a grid. There is a notion of the "current index", but no notion of an anchor or cursor that would not be an index of an element. Further, the current index's update rules are questionable at times: selecting a range sets the current index at the end of the range, inviting the programmer to implement a selection feature that treats range selections that grow downwards or right differently from those that grow upwards or left. There is no notion of a selection path that could support lasso-like selection tools, no notion of an active domain, and no support for implementing an undo operation.

Qt's [16] QListView, QTableView, and QTreeView classes provide a pre-packaged selection feature. The QItemSelectionModel class serves the same role as JavaFX's MultipleSelectionModel, and has most of the same limitations. QItemSelectionModel, however, supports deselecting regions (rectangular areas in a grid). It also separates the "current selection" from the permanent (baked in our terminology) selection, which is useful for implementing a shift-click feature that remembers the selection state under the active domain, as in our model.

The story with Web GUI frameworks is similar. Taking one popular example, jQuery UI library [15] provides "Selectable" as one of its user interface interaction widgets. Its multi-selection feature of DOM elements supports rectangular rubber band selection and toggling with command-click. Shift has no function and keyboard selection is not supported. Command-click on a selected element is quite surprising: the element is first deselected, but even the smallest mouse move reselects the just deselected element, and starts a rubber band selection. The library is of little use when the pre-packaged selection is not adequate.

To summarize, today's GUI frameworks provide some widgets and views with multi-selection, but they do not provide a reusable multi-selection feature. At best they provide a set of utility classes that can be useful building blocks in implementing such a feature. They leave arguably the most difficult tasks to the programmer: gaining a thorough understanding of how multi-selection should work and mapping that understanding to an implementation.

That our work models multi-selection formally differentiates it further from prior works on multi-selection. More generally, modeling user interaction and interfaces formally has not found a particularly strong foothold in practical GUI programming. This point has been made by many over the years [7, 17, 5], before introducing new techniques aimed at changing the status quo. The result is a diverse set of approaches, none in the mainstream. We do not propose a new approach for formally modeling user interfaces, but instead apply common tools and notations from the domain of programming languages for defining the operational semantics of an abstract "multi-selection machine".

Developing the formal model in the manner we did certainly required effort and iterations, but the effort is justified by the prospect of reusing the result widely and by the resulting clear understanding of the modeled feature. We suspect that there are other programming challenges in user interfaces that would benefit from an analogous formal inspection; some prior work on those lines exists. Krishnamurthi et al. [9] model the user's interaction with a web browser communicating with a server. They recognize that defects manifest frequently when users mix the browser's page navigation tools with those provided by a web application. Practical evidence shows that coordinating web interaction is clearly complex enough that programmers do not consistently implement it correctly. Analogously to multi-selection, a formal model clears the confusion and provides a precise specification. Concretely, their "language transition relation" serves a similar role than our selection language.

As another example of formally modeling oft-occurring user interface components, Zhang [17] defines models for buttons, menus, textual input fields, etc. combining modeling interaction with finite state machines and algebraic specification. Formal models of user

interaction are often finite state machines and use notation designed for their concise representation (See Bowen et al. [5] for a survey of many techniques). Our selection language can also be seen as a definition of a state machine, but the abstract machine view we chose is more useful; the state changes are conveniently captured by changes to the selection state tuple and enables also algebraic reasoning (e.g., for developing optimizations).

The early PIE-models [6] would offer a setup for inspecting properties such as the reachability of all selection states, and aspects of undo, but these questions in our model are rather straightforward.

Finally, we note that there is a wealth of CHI literature about innovative means of selecting groups of elements with different kinds of pointing devices. A recent article by Strothoff et al. [14] contains a summary and categorization of such works. The connection to our work is tangential; our contribution is not in novel means of selection, but rather in how to cost-effectively and correctly implement the widely known and established multi-selection features. Perhaps our work could help in designing and implementing novel selection schemes.

7 Conclusion

Multi-selection is a mundane feature, yet evidently too complex to design well and implement correctly as part of a typical application development project. Practically every implementation manifests at least minor problems, some are flawed in major ways, and all are different. Even the most established applications do not stand up to close scrutiny.

We would like to see this change. Users should be able to rely on a full-fledged, correct, and uniform multi-selection feature in all contexts where selecting many elements would be useful. Consequently, this paper describes the abstractions and software architecture that makes multi-selection reusable. With a reusable multi-selection library, practically any kind of collection of elements can be made selectable with little programming effort, and every context where selection is supported will provide the same rich set of selection commands.

This article is not meant to discourage innovation and experimentation in how best to interact with collections of elements. Redesigning and reimplementing the same features many times over with predictably unpredictable outcomes, however, is an effort misplaced.

References

1 Apple Computer, Inc. Lisa user interface starndards. Internal technical document, September 1980. URL: `www.guidebookgallery.org/articles/lisauserinterfacestandards`.

2 Apple Computer, Inc. *Macintosh Human Interface Guidelines*. Addison-Wesley, USA, 1992.

3 Apple Inc. Mac Developer Library, 2015. URL: `developer.apple.com/library/mac/navigation`.

4 Apple Inc. OS X El Capitan: Select items, 2015. URL: `support.apple.com/kb/PH21888`.

5 Judy Bowen and Steve Reeves. Formal models for user interface design artefacts. *Innovations in Systems and Software Engineering*, 4(2):125–141, 2008.

6 A. J. Dix and C. Runciman. Abstract models of interactive systems. In *HCI'85: People and Computers: Designing the Interface*, pages 13–22. Cambridge University Press, 1985.

7 Michael D. Harrison and H. Thimbleby, editors. *Formal Methods in Human Computer Interaction*. Cambridge University Press, 1990. Chapter 1.

8 John Karat, James E. McDonald, and Matthew P. Anderson. A comparison of menu selection techniques: Touch panel, mouse and keyboard. *International Journal of Man-Machine Studies*, 25(1):73–88, 1986.

9 Shriram Krishnamurthi, Robert Bruce Findler, Paul Graunke, and Matthias Felleisen. *Interactive Computation: The New Paradigm*, chapter Modeling Web Interactions and Errors, pages 255–275. Springer, Berlin, Heidelberg, 2006. doi:10.1007/3-540-34874-3_11.

10 Jonathan Lazar, Adam Jones, Mary Hackley, and Ben Shneiderman. Severity and impact of computer user frustration: A comparison of student and workplace users. *Interacting with Computers*, 18(2):187–207, March 2006. doi:10.1016/j.intcom.2005.06.001.

11 Microsoft. Select multiple files or folders. MS Windows support topic. Accessed: 2015-11-31. URL: windows.microsoft.com/en-us/windows7/select-multiple-files-or-folders.

12 Microsoft. *Windows Interface Guidelines for Software Design*. Microsoft Press, Redmond, WA, USA, 1st edition, 1995.

13 Oracle. JavaFX, MultipleSelectionModel. Accessed: 2015-12-07. URL: docs.oracle.com/javase/8/javafx/api/toc.htm.

14 Sven Strothoff, Wolfgang Stuerzlinger, and Klaus Hinrichs. Pins 'n' touches: An interface for tagging and editing complex groups. In *Proc. of the 2015 Int. Conf. on Interactive Tabletops & Surfaces*, ITS '15, pages 191–200, New York, NY, USA, 2015. ACM.

15 The jQuery Foundation. jQuery user interface. Accessed: 2015-12-03. URL: jqueryui.com.

16 The Qt Company. Qt Documentation. Accessed: 2015-12-03. URL: doc.qt.io.

17 Weishi Zhang. *Formal description and development of graphical user interfaces*. Utz, 1996.

Program Tailoring: Slicing by Sequential Criteria*

Yue Li[†1], Tian Tan[†2], Yifei Zhang[3], and Jingling Xue[4]

1 **School of Computer Science and Engineering, UNSW Australia**
yueli@cse.unsw.edu.au
2 **School of Computer Science and Engineering, UNSW Australia**
tiantan@cse.unsw.edu.au
3 **School of Computer Science and Engineering, UNSW Australia**
yzhang@cse.unsw.edu.au
4 **School of Computer Science and Engineering, UNSW Australia**
jingling@cse.unsw.edu.au

Abstract

Protocol and typestate analyses often report some sequences of statements ending at a program point P that needs to be scrutinized, since P may be erroneous or imprecisely analyzed. Program slicing focuses only on the behavior at P by computing a slice of the program affecting the values at P. In this paper, we propose to restrict our attention to the subset of that behavior at P affected by one or several statement sequences, called a *sequential criterion (SC)*. By leveraging the ordering information in a *SC*, e.g., the temporal order in a few valid/invalid API method invocation sequences, we introduce a new technique, *program tailoring*, to compute a tailored program that comprises the statements in all possible execution paths passing through at least one sequence in *SC* in the given order. With a prototyping implementation, TAILOR, we show why tailoring is practically useful by conducting two case studies on seven large real-world Java applications. For program debugging and understanding, TAILOR can complement program slicing by removing *SC*-irrelevant statements. For program analysis, TAILOR can enable a pointer analysis, which is unscalable to a program, to perform a more focused and therefore potentially scalable analysis to its *SC*-relevant parts containing hard language features such as reflection.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages - Program Analysis, D.2.5 Testing and Debugging - Code inspections and Debugging aids

Keywords and phrases Program Slicing, Program Analysis, API Protocol Analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.15

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.2.1.8

1 Introduction

Program slicing, supported by industry-strength tools, such as WALA [52] and CodeSurfer [18], has found many diverse applications, such as program debugging, comprehension, analysis, testing, verification and optimization [8, 20, 43, 49]. Given a *slicing criterion* consisting of a program point P and several variables used at P [53], program slicing computes a slice of the program that may affect their values at P in terms of data and control dependences. In the

* This work is supported by ARC grants, DP130101970 and DP150102109.
† These authors contributed equally to this work.

30th European Conference on Object-Oriented Programming (ECOOP 2016).
Editors: Shriram Krishnamurthi and Benjamin S. Lerner; Article No. 15; pp. 15:1–15:27


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

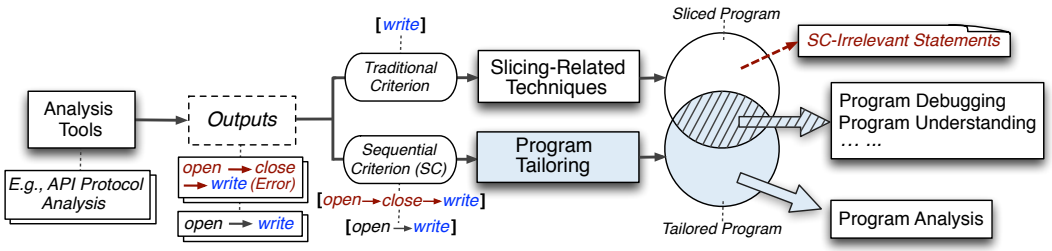


Figure 1 Program tailoring with some of its potential applications highlighted.

past three decades, several variations on this theme of program slicing have been proposed, including static vs. dynamic, backward vs. forward, and closure vs. executable [43, 49].

In practice, API protocol analysis [7, 37] and typestate analysis [9, 16, 34] often report some statement sequences ending at a program point P that needs to be scrutinized, since P may be erroneous or imprecisely analyzed. Each sequence can represent a valid or invalid API usage call sequence. Such protocol specifications or violations can also be provided manually or mined automatically [1, 3, 17, 36, 42, 60]. As such analyses are either conservative or unsound, the temporal order specified in a sequence may or may not be feasible. However, program slicing focuses on P by ignoring this order, which is often essential for analyzing P.

As illustrated in Figure 1, we introduce a new technique, *program tailoring* to reap additional benefits missed by program slicing at a point P (e.g., `file.write()`) by leveraging the temporal order specified in a new criterion, called a *sequential criterion* (*SC*) for P. A SC_P consists of one or several statement sequences ending at P, with each representing, e.g., a valid API call sequence like `file.open()` → `file.write()` or an invalid API call sequence like `file.open()` → `file.close()` → `file.write()`. In what follows, we will drop P from SC_P when the context is clear or when we are not interested in it. Given a SC_P, tailoring aims to obtain a tailored program, denoted $\mathcal{T}(SC_P)$, that comprises the statements in all the possible execution paths passing through at least one statement sequence in SC_P in the given order. By construction, all data and control dependences needed for understanding the behavior at P affected by SC_P are included. Any statement that is not in $\mathcal{T}(SC_P)$ is irrelevant to SC_P, i.e., SC_P-*irrelevant*. For illustration purposes, we write $\mathcal{S}(P)$ to represent the (backward) slice affecting P obtained by program slicing. Note that slicing all the points in SC_P independently still fails to capture their ordering constraint (and is unscalable, too).

Like slicing, tailoring enables software developers or client applications to inspect only the interesting parts of a program. Unlike slicing, which focuses on understanding the program behavior at P, tailoring restricts our attention to the subset of that behavior affected by SC_P only. Due to incompatible criteria used, tailoring can be used either as a complementary technique to slicing or in cases where slicing is ineffective, as discussed below.

1.1 Goals and Motivations

Given a SC_P, we have developed a prototyping implementation of program tailoring, denoted Tailor, for Java programs, with the following three goals in mind:

Precision Tailor is designed to sharpen the precision of many client applications, as highlighted in Figure 1, by exploiting the temporal order in a SC_P. One significant class of client applications includes many slicing-related techniques, such as thin slicing [47], program chopping [22] and value slicing [26]. For a given program, $\mathcal{T}(SC_P)$ is shown as the blue circle and $\mathcal{S}(P)$ as the white circle. The statements in $\mathcal{I}(SC_P) = \mathcal{S}(P) - \mathcal{S}(P) \cap \mathcal{T}(SC_P)$ are SC_P-irrelevant and can thus be pruned away to facilitate program debugging

and understanding by a human. If $\mathcal{I}(SC_P) = \varnothing$, then Tailor is ineffective at P but no harm is done. If $\mathcal{I}(SC_P) \neq \varnothing$, then Tailor can make slicing more precise, by leveraging the otherwise wasted SC_P information that is widely available. In Section 6.1, we show that Tailor can improve the precision of thin slicing [47], a state-of-the-art practical but unsound slicing technique, for Java programs. In Section 6.2, we show that Tailor can enable a sophisticated pointer analysis for Java, S-2Obj [24], which is unscalable for a program, to perform a focused analysis on its SC-relevant parts containing hard language features such as reflection, where existing slicing techniques are ineffective.

Scalability Tailor is designed to work efficiently for large object-oriented programs, for which traditional slicing [21] is unscalable (even with industry-strength implementations [18,52]), with the key bottleneck coming from handling of the heap [47]. Like any slicing tool, Tailor is not always scalable. However, Tailor is designed to scale significantly better than traditional slicing, as Tailor avoids handling of the heap by reasoning about essentially the (un)reachability of a statement towards the statement sequences in a SC_P.

Soundiness Tailor is designed to be a practical tool to facilitate programming debugging and understanding as well as program analysis (among others) for large object-oriented programs. In this setting, any sound static analysis would be either unscalable or imprecise due to the presence of many hard language features, such as native code, dynamic class loading, reflection and multi-threading [30]. Therefore, Tailor is designed to be sound whenever a sound graph representation is available for capturing all the control flows in a (single- or multi-threaded) program. In other words, Tailor is always sound with respect to part of the program behavior modeled. According to [30], "a soundy analysis aims to be as sound as possible without excessively compromising precision and/or scalability." Therefore, Tailor represents one such soundy analysis.

1.2 Challenges and Solutions

We examine some challenges faced in achieving our three goals and describe our solutions:

Precision/Scalability Tradeoffs How do we compute $\mathcal{T}(SC)$ efficiently and precisely for large object-oriented programs? Due to exponential blowup of program paths, both DFS and BFS are out of question. We formulate the problem of computing $\mathcal{T}(SC)$ by solving an IFDS (Interprocedural Finite Distributive Subset) data-flow analysis problem [38], efficiently on its interprocedural control-flow graph (ICFG) representation of the program. To avoid unrealizable paths with mismatched calls and returns, our analysis must be (fully) context-sensitive in order to achieve useful precision for Java programs. Otherwise, many unrealizable paths, which go through the constructor of `java.lang.Object`, cannot be filtered out, causing $\mathcal{T}(SC)$ to be severely over-approximated. However, distinguishing calling contexts fully with call strings, object-sensitivity [33], and method cloning [54] are known to be unscalable for large programs [12, 44]. We achieve (full) context sensitivity by solving a CFL-reachability problem over a balanced-parentheses language by matching call and return edges also in the IFDS framework as described in [38].

How do we ensure that Tailor works effectively for a SC of any given length, which is defined to be the number of statements in its longest statement sequence? In general, the longer a SC is, the more SC-irrelevant statements will be removed. We propose to lengthen any given SC by leveraging the concept of object-sensitivity [33, 44] developed by the pointer analysis community for Java. As a result, some infeasible paths that would otherwise be introduced are avoided. We will also try to avoid making SC extensions that make our analysis run longer but contribute nothing to precision improvement.

Soundiness How do we make TAILOR as soundly as possible? We decompose the problem of tailoring a program for a given SC into two sub-problems: (1) building an ICFG, G_{ICFG}, for the program and (2) computing $\mathcal{T}(SC)$ from G_{ICFG}. TAILOR is sound if G_{ICFG} is sound (representation of all control flows in the program). In fact, TAILOR is designed to be practically useful for analyzing the program behavior modeled by G_{ICFG} even if G_{ICFG} is unsound. This paper solves (2) while resorting to the state-of-the-art for (1).

1.3 Contributions

- We introduce program tailoring, a new technique for trimming a program based on *SCs*, which are widely available from other analyses but never exploited by program slicing.
- We describe how to extend a given SC for object-oriented programs in order to improve the precision of program tailoring (by making tailored programs smaller).
- We formulate the problem of computing a tailored program as one of solving a data-flow problem efficiently in the IFDS framework. TAILOR, which is implemented in SOOT [51], is released as an open-source tool at `http://www.cse.unsw.edu.au/~corg/tailor`.
- We describe two case studies to demonstrate why TAILOR is practically useful on a set of seven large real-world Java programs, by (1) assisting program slicing with program debugging and understanding tasks, and (2) enabling a focused pointer analysis on the parts of a program containing hard language features such as reflection.

2 A Motivating Example

We use an example to describe how TAILOR can assist slicing tools to simplify program debugging and understanding tasks through exploiting the temporal ordering information in a given SC that is otherwise ignored by program slicing. In Section 6.2, we provide additional motivations on why TAILOR can be useful in simplifying program analysis tasks.

Large object-oriented programs are very difficult to debug and understand, due to the pervasive use of heap-allocated data, nested data structures, and large libraries with complex dependences and configurations. Tracing the flow of values via multiple levels of pointer indirection through the heap across many classes in both the application and libraries is unworkable. A practical tool is needed to pinpoint relevant statements for the task at hand.

Our Java example is given in Figure 2. The `Driver` class is used to create and initialize a `Driver` object according to some user input (lines 33 – 36) or by default (lines 37 – 41). Then the corresponding initialization information stored in `info` is dumped to a log at line 42.

This example has a typical error found in Java programs caused by ignoring the fact that closing a wrapper file handler will also close its internal file handler. The internal file handler, `fw` is passed as an argument at line 6 and assigned to `out` at line 25. Later, closing its wrapper file handler, `bw`, at line 9 will also close `out` (i.e., `fw`) at line 27. Then any further access to a closed `fw` (e.g., at line 12) will trigger a runtime exception at line 20.

Now we use a static typestate analysis tool CLARA [9] to analyze this program. Some typestate specifications regarding file operations used will include "accessing a closed file leads to an error state". For our example, CLARA produces an error report shown on the left, marking line 12 as a "*potential point of failure*", together with a sequence of two method calls leading to line 12. We have one $SC_{\text{line 12}}$: line 2 → line 27 → line 12. As static analyses like CLARA are either conservative or unsound, there may or may not be an error at line 12. Now our debugging task begins.

Potential Point of Failure :
Statement: fw.write(info) at line 12
Related Program Points :
Statement: out.close() at line 27
Statement: new FileWriter(...) at line 2

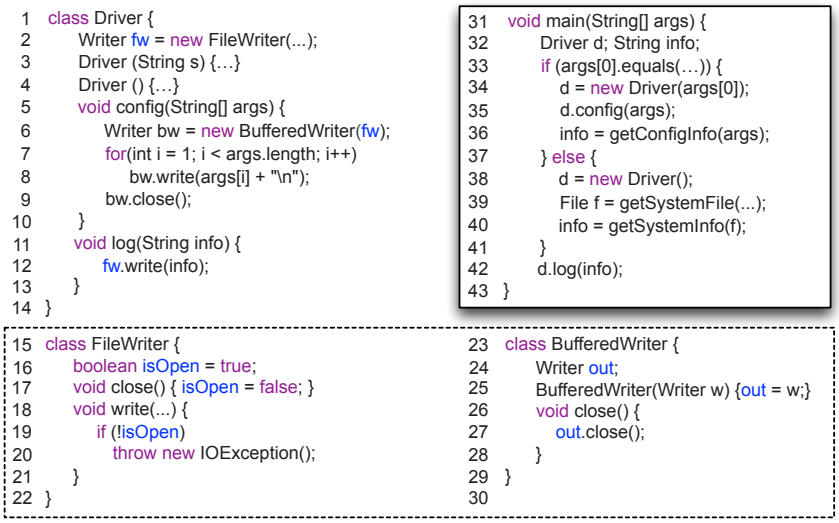


Figure 2 An example showing how TAILOR removes $SC_{\text{line }12}$-irrelevant statements.

The error at line 12 happens only when a `Driver` object is created in the `if` branch of `main()`. Therefore, a tool that instructs the developer to examine this `if` branch only would enable its cause to be identified quickly. In contrast, marking some lines in its matching `else` branch as also being relevant can increase human analysis effort significantly (especially if the developer has to trace the flow of values across many nested heap structures).

Traditional Slicing. For large object-oriented programs, traditional (sound) slicing [21, 53] is unscalable or yields slices that are too large for human consumption [47]. Given a virtual call `fw.write(info)` at line 12, the set of variables of interest consists of (1) the receiver reference and the arguments of the call [52], and (2) some relevant variables selected manually or recognized automatically [2, 13]. In our example, these variables form the set {`fw`, `info`, `fw.isOpen`}. Then, a (backward) slice that affects their values comprises all the statements except lines 7 – 8 and 19 – 20. This slice, which includes everything in `main()`, contains too many statements that are not all directly relevant to the task at line 12.

Thin Slicing. Thin slicing [47] is introduced to facilitate program debugging and understanding for object-oriented programs by trading soundness for scalability and (direct) relevance. All control dependences and all base pointer data dependences are excluded. Given a point of interest, thin slicing includes only so-called *producer statements* that affect directly the values at the point. Statements that serve to explain why producer statements affect the point are ignored. For example, given $x = p.f$ and $q.f = y$, where p and q may be aliased, $q.f = y$ is a producer statement for $x = p.f$, because there may be a direct value flow from y to x. All statements that help explain or establish why p and q are aliases are ignored.

If we adopt the same slicing criterion at line 12 as above, thin slicing will include only seven statements at lines 2, 12, 16, 17, 36, 39 and 40. Compared with the traditional slice obtained, this smaller slice still retains line 17, a statement for explaining an immediate cause of the error at line 12. However, two $SC_{\text{line }12}$-irrelevant statements at lines 39 – 40 (in the `else` branch of `main()`) are also present, which can cost human analysis effort unnecessarily.

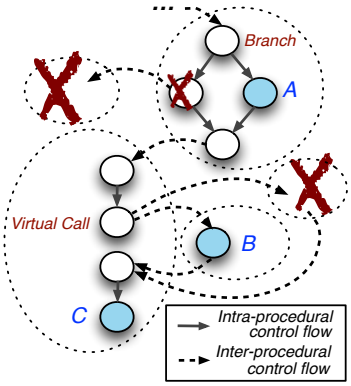


Figure 3 Leveraging the ordering information $SC : A \rightarrow B \rightarrow C$ to trim irrelevant statements away.

Program Tailoring. Given $SC_{\text{line 12}}$: line 2→line 27→line 12, TAILOR produces a tailored program consisting of all the statements in all execution paths passing through lines 2, 27 and 12 in that order. As line 27 is only reachable from line 9, which resides in the `config` method invoked at line 35, the tailored program includes all the lines in the example *except lines 37 – 41 that appear in the* `else` *branch of* `main()` and lines 19 – 20.

Let us revisit the two slices obtained above, the traditional slice, P_{trad} and the thin slice, P_{thin}. Let our tailored program be identified as P_{tal}. Despite $|\text{P}_{\text{thin}}| < |\text{P}_{\text{tal}}| < |\text{P}_{\text{trad}}|$, tailoring brings several benefits, obtained from exploiting the temporal order of invocation sequences in $SC_{\text{line 12}}$. First, P_{tal} is the only one that includes nothing from the `else` branch of `main()`, revealing more clearly to a human debugger that the potential error at line 12 is triggered by a `Driver` object created in the `if` branch of `main()` ("according to some user input") rather than in its matching `else` branch ("according to some default configuration"). Such contextual information enables the developer to locate the cause for the error at line 12 more quickly. In the case of P_{trad} and P_{thin}, the developer may end up wasting a lot of analysis effort on navigating through a lot of $SC_{\text{line 12}}$-irrelevant code, highlighted by `getSystemInfo()` and `getSystemFile()`, in the `else` branch. Second, any statement that is not included in P_{tal} is $SC_{\text{line 12}}$-irrelevant for understanding the SC-specific behavior at line 12. Thus, TAILOR can make, e.g., thin slicing more effective, by removing the $SC_{\text{line 12}}$-irrelevant statements $\text{P}_{\text{thin}} - \text{P}_{\text{thin}} \cap \text{P}_{\text{tal}}$, i.e., lines 39 and 40, from P_{thin}. Then P_{thin} is down to only five statements at lines 2, 12, 16, 17 and 36. Starting from line 17 (an immediate cause for the error at line 12), the developer can trace the flow of `isOpen` backwards to find the original cause. Finally, P_{tal} includes all data and control dependences reaching line 12 affected by $SC_{\text{line 12}}$, enabling it to be analyzed further by other analyses, e.g., a pointer analysis, as will be discussed in Section 6. However, P_{trad} and P_{thin} will not be applicable since P_{trad} is unobtainable scalably for large programs and P_{thin} is unsound.

Note that P_{tal} still contains lines 7 – 8 that do not affect $SC_{\text{line 12}}$. Removing *all* such irrelevant statements for large programs may be neither necessary (due to the first two points made earlier) nor practical, as a sound slicing tool would be unscalable. Thus, we have designed TAILOR based on the precision/scalability tradeoffs described in Section 1.

Figure 3 recaps our insight behind tailoring. Given a $SC : A \rightarrow B \rightarrow C$, we focus on the behavior at C affected by a sequential execution of A, B and C. If one point in SC is reached from only one branch (e.g., the one containing A) of a multi-way branching statement, then the statements in the other branches are SC-irrelevant and can be trimmed away (✗). This

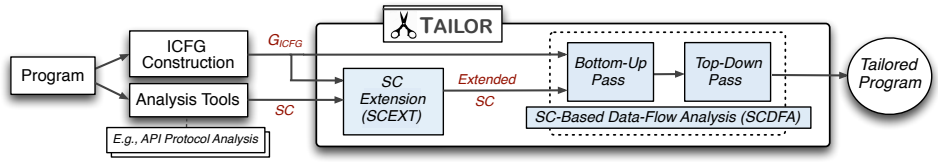


Figure 4 Overview of TAILOR (with all the components implemented in this paper in blue).

is particularly suitable for object-oriented languages, since a virtual call site behaves as a multi-way branch switching to its target methods. For example, B can be regarded as residing in a target method invoked at the marked virtual call on a receiver object created only at the allocation site at A. Thus, the other target methods unreachable to C are trimmed away (✘).

3 Methodology

Figure 4 gives an overview of TAILOR, with all the components implemented in this paper highlighted in blue. Given a program, we rely on the state-of-the-art (shown as "ICFG Construction") to build an inter-procedural control flow graph (ICFG), denoted G_{ICFG}, to represent all the possible control flows in the program. A SC is simply a set of one or more statement sequences ending at the same statement, with all statements identified by their line numbers, i.e., program points only. The *length* of a SC is the number of statements in its longest sequence. *SCs* can be deduced from the results returned by many analysis tools such as API protocol analysis [7, 37] and typestate analysis [9, 16, 34]. For example, a typestate analysis may report a potential error at line C, `f.write()`, together with two invocation sequences of related methods, A : `f.open()` $\rightarrow$ $B1$: `f1.close()` and A : `f.open()` $\rightarrow$ $B2$: `f2.close()`, leading to line C. Therefore, we may choose to tailor the program at C to investigate its behavior affected by $SC = \{A \rightarrow B1 \rightarrow C, A \rightarrow B2 \rightarrow C\}$.

Given a SC, a tailored program, $\mathcal{T}(SC)$, consists of the statements on all possible execution paths in G_{ICFG} passing through at least one statement sequence in SC. By exploiting the temporal ordering information in SC, it is possible to scale tailoring significantly better than slicing for large object-oriented programs and makes it a practically useful technique.

TAILOR is sound as it computes $\mathcal{T}(SC)$ over-approximately with respect to G_{ICFG}. In contrast, traditional (sound) slicing [21] is unscalable when G_{ICFG} is large and thin slicing [47] is unsound for G_{ICFG} (as it is designed for program debugging and understanding only).

TAILOR computes $\mathcal{T}(SC)$ in two stages, SC Extension and SC-based Data-Flow Analysis. In the first stage (Section 3.2), we exploit "branch correlations" in object-oriented programs to lengthen a given SC in order to avoid some infeasible paths that would otherwise be introduced in the second stage. In the second stage (Section 3.1), we compute $\mathcal{T}(SC)$ by solving a data-flow problem in order to avoid unrealizable paths efficiently. This design allows TAILOR to achieve good precision and scale well to large object-oriented programs.

3.1 SC-Based Data-Flow Analysis

We compute $\mathcal{T}(SC)$ by solving flow- and context-sensitively an IFDS (Interprocedural Finite Distributive Subset) data-flow problem [38], efficiently on G_{ICFG}, via graph reachability. This formulation of our SC-based data-flow analysis, denoted SCDFA, is significant for three reasons. (1) With flow-sensitivity, SCDFA can filter out imprecisely ordered statement sequences in a SC, as many static analyses from which *SCs* are deduced are flow-insensitive

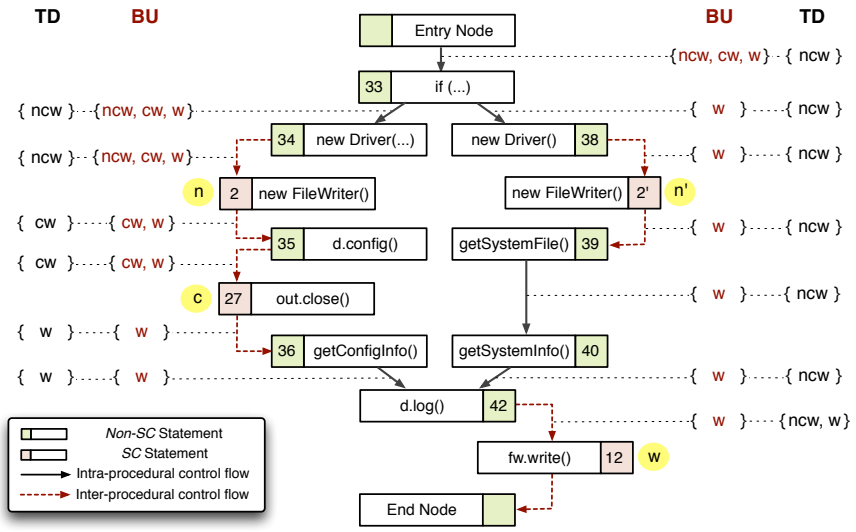


Figure 5 A simplified G_{ICFG} of the program given in Figure 2 for illustrating the bottom-up (BU) and top-down (TD) passes of SCDFA with $SC_w = \{n{\rightarrow}c{\rightarrow}w, n'{\rightarrow}c{\rightarrow}w\}$.

in order to be scalable. Conversely, precisely ordered statement sequences in SC also enable SCDFA to filter out irrelevant control flows in tailoring a program. This mutually beneficial process improves the precision of both parts, therefore avoiding the unnecessary complexity faced for solving both as one problem. (2) With context-sensitivity, we avoid introducing unrealizable paths with mismatched calls and returns, which is critical for achieving precision in computing $\mathcal{T}(SC)$. (3) With an IFDS formulation, SCDFA can scale well to reasonably large object-oriented programs. In particular, (full) context-sensitivity can be realized more efficiently by solving a CFL-reachability problem over a simplified balanced-parentheses language. As an IFDS problem, SCDFA has a time complexity of $O(ED^3)$, where E is the number of edges in G_{ICFG} and D is the size of the set of SC-related data-flow facts used (i.e., the set of suffixes of sequences in SC, as will be clear below and defined in Section 4).

SCDFA starts with a bottom-up pass (BU) and finishes with a top-down pass (TD), with both performed (fully) context-sensitively. By using the example program from Figure 2, we first explain the functionalities of two passes and then examine briefly how context-sensitivity is realized efficiently in the IFDS framework. Suppose we are given a SC as line 2 : `fw = new FileWriter()` $\rightarrow$ line 27 : `out.close()` $\rightarrow$ line 12 : `fw.write()`. For convenience, we write it as $SC : n{\rightarrow}c{\rightarrow}w$. In its G_{ICFG}, the allocation site for `FileWriter` at line 2 is replicated in the two constructors of `Driver`. We identify the one in the single-arg constructor as line 2 (denoted by n) and the one in the non-arg constructor as line 2′ (denoted by n'). As a result, we finally have a two-sequence $SC_w = \{n{\rightarrow}c{\rightarrow}w, n'{\rightarrow}c{\rightarrow}w\}$.

***BU* and *TD* Passes.** Both passes operate on G_{ICFG}, as shown in Figure 5, for our example. As in any data-flow analysis, all control flow edges in G_{ICFG} are treated non-deterministically executable. Let ENTRY be the entry node of G_{ICFG} and EXIT the node marking the point of interest w at line 12 in SC_w. Conceptually, if a sequence $S \in SC_w$, which is ncw or $n'cw$, lies on a path, then BU must see a suffix of S at every node n on the path backwards from EXIT and TD must see the corresponding prefix of S at n forwards from ENTRY.

BU computes a global property, PANTI, for every node n in G_{ICFG}, backwards against the

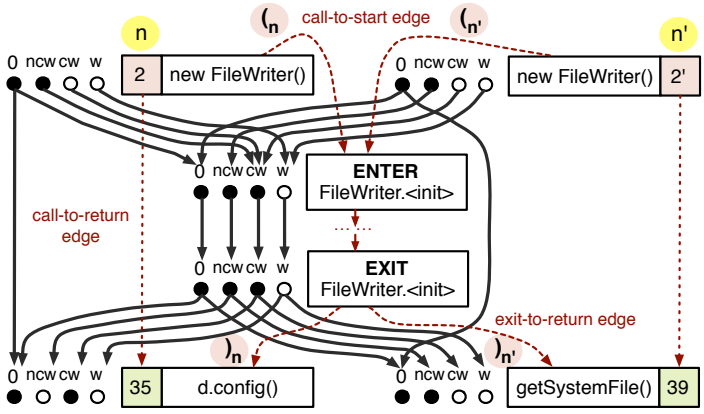


Figure 6 Context-sensitivity via CFL-reachability.

control flow, starting at EXIT. $\mathsf{PANTI}^{\text{out}}(n)$ represents the set of suffixes of some sequences in SC_w that are partially anticipable at the entry of n, i.e., appear on some outgoing path of n ending at EXIT. In our example, $\mathsf{PANTI}^{\text{out}}(\text{ENTRY}) = \{ncw, cw, w\}$. As ncw is partially anticipable (but $n'cw$ is not) at ENTRY, G_{ICFG} contains some paths passing through ncw.

TD computes a global property, PAVAIL, for every node n in G_{ICFG}, forwards along the control flow, starting from ENTRY. $\mathsf{PAVAIL}^{\text{in}}(n)$ specifies the set of suffixes of some sequences in $\mathsf{PANTI}^{\text{out}}(\text{ENTRY})$ to represent implicitly (for efficiency reasons) the fact that their corresponding prefixes are partially available at the entry of n, i.e., appear on some incoming paths of n starting from ENTRY. For example, $\mathsf{PAVAIL}^{\text{in}}(12) = \{ncw, w\}$, indicating that the prefixes ϵ and nc of ncw are partially available at the entry of node 12.

For this example, a node n is included in the tailored program if $\mathsf{PAVAIL}^{\text{in}}(n) \cap \mathsf{PANTI}^{\text{out}}(n) \neq \varnothing$, since a suffix $s \in \mathsf{PAVAIL}^{\text{in}}(n) \cap \mathsf{PANTI}^{\text{out}}(n)$ is partially anticipable at the entry of n and some sequence in SC_w with s removed is partially available at the entry of n. In our example, the tailored program consists of all the lines except lines 38 – 40.

Context Sensitivity. Figure 6 illustrates as an example how the *TD* pass shown in Figure 5 is performed for `FileWriter()` context-sensitively by solving a CFL-reachability-based balanced-parentheses problem, efficiently in the IFDS framework. For technical details, we refer to [38]. There are four data-flow facts, ncw, cw, w, and 0 (for the empty set). There are two call sites to `FileWriter()` at lines 2 and 2′, which are identified as nodes n and n', respectively, with their call-to-start and exit-to-return edges labeled as $(_n$, $)_n$, $(_{n'}$ and $)_{n'}$ appropriately. ENTER and EXIT are the start and exit nodes of `FileWriter()`, whose CFG is irrelevant and thus elided. In SCDFA, the call-to-return edge always serves as a kill edge (to stop the data-flow facts from bypassing a callee). For each node n, $\mathsf{PAVAIL}^{\text{in}}(n)$ is the set of facts, i.e., suffixes of ncw shown in black dots. According to Figure 5, $\mathsf{PAVAIL}^{\text{in}}(2) = \mathsf{PAVAIL}^{\text{in}}(2') = \{ncw\}$. If `FileWriter()` is entered from $(_n$ and exited from $)_n$, then $\mathsf{PAVAIL}^{\text{in}}(\text{ENTER}) = \mathsf{PAVAIL}^{\text{in}}(\text{EXIT}) = \{cw\}$. Hence, $\mathsf{PAVAIL}^{\text{in}}(35) = \{cw\}$. However, if `FileWriter()` is entered from $(_{n'}$ and exited from $)_{n'}$, then $\mathsf{PAVAIL}^{\text{in}}(\text{ENTER}) = \mathsf{PAVAIL}^{\text{in}}(\text{EXIT}) = \{ncw\}$. Hence, $\mathsf{PAVAIL}^{\text{in}}(39) = \{ncw\}$.

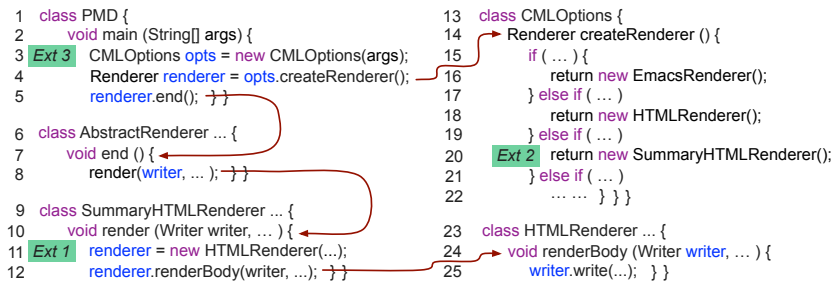


Figure 7 A code snippet from PMD for illustrating SCEXT in a program understanding task.

3.2 SC Extension

TAILOR is designed to work effectively with any SC. While an API specification mining tool [17] may generate SCs longer than 10, an assertion verifier may produce only single-point SCs. In general, the longer a SC is, the more SC-irrelevant statements will be eliminated.

To improve the precision of SCDFA, we perform first a SC extension pass, denoted SCEXT, as shown in Figure 4. Given a sequence $S \in SC$, with F being its first point, we look for a set of n extension points $E_1, \cdots, E_n$, that collectively dominate F, such that any program execution that passes through F must pass through one E_i. We do so effectively by leveraging the concept of object-sensitivity [33,44] developed by the pointer analysis community for Java. For F, its extension points are chosen as the object allocation sites used for representing the object-sensitive calling contexts for the method containing F. As a result, some infeasible paths are avoided by exploiting branch correlations in object-oriented programs (Figure 3).

To make SC extensions, we use the points-to information obtained by, e.g., the pointer analysis performed earlier for building G_{ICFG}. Consider an ICFG fragment discussed earlier in Figure 3. Suppose we are given $SC : B \rightarrow C$ such that B resides in a target method m for the virtual call site shown and m is only invokable on the objects created at A, which represents an object allocation site. Then we can prepend A to SC to obtain $ESC : A \rightarrow B \rightarrow C$. According to SCDFA, $\mathcal{T}(SC)$ will include both branches of "Branch" shown in Figure 3 since both reach SC but $\mathcal{T}(ESC)$ will include only the branch containing A as the other (infeasible) branch does not reach ESC. However, lengthening a SC increases the number of SC-related data-flow facts used (Figure 5). So a precision/scalability tradeoff needs to be made.

We use a real program understanding task to show why SCEXT is useful for real code. PMD is a static analyzer for analyzing Java programs and can print a range of source code flaws in different formats such as Emacs, CSV, and HTML. In this task, we want to understand how PMD renders its outputs in the commonly used HTML format. Figure 7 shows the simplified code. The only knowledge we have initially is that the `HTMLRenderer` class is responsible for writing to the file at line 25, but how it is done is unknown. At this stage, we have a single-point, $SC_{\text{line 25}}$: line 25, representing just the `write` statement at line 25.

If we apply SCDFA to $SC_{\text{line 25}}$, $\mathcal{T}(SC_{\text{line 25}})$ will include all the lines in the code given. Below we show how to extend this to $ESC_{\text{line 25}}$: line 20 $\rightarrow$ line 11 $\rightarrow$ line 25 object-sensitively [33, 44]. If we apply SCDFA to $ESC_{\text{line 25}}$, $\mathcal{T}(ESC_{\text{line 25}})$ will now be smaller, consisting of all the lines except lines 16, 18 and 22 in method `createRenderer()`. In other words, all the branches except the one enclosing line 20 in method `createRenderer()` are infeasible for $SC_{\text{line 25}}$, according to the points-to information provided for this program.

Let us see how SCEXT works in growing $SC_{\text{line 25}}$ to become $ESC_{\text{line 25}}$. Initially, SC_{line25} has one statement at line 25, which resides in method `renderBody()`. The (abstract) receiver

object pointed to by `renderer` at line 12 is allocated only at the allocation site at line 11 and is considered as the calling context for line 25 in an object-sensitive pointer analysis. There is another allocation site at line 18 for the same type, `HTMLRenderer`, but this is not considered, since the abstract object created at line 18 does not flow to line 12 (according to the points-to information available). At this stage, we have $SC_{\text{line 25}}$: line 11 $\rightarrow$ line 25.

Similarly, we now look for the allocation sites that can be used as the calling contexts for line 11 contained in method `render()` at line 10, which is called from `this.render()` at line 8 in method `end()` defined in the `AbstractRenderer` class. Thus, the receiver objects on which method `render()` (line 10) is invoked are the ones pointed to by `renderer` at line 5. These objects are returned from the call to `createRenderer()` at line 4. Thus, `renderer` points to the objects created by the allocation sites in all different branches in `createRenderer()`. According to the points-to information, the target method `end()` invoked at the virtual call site at line 5 can only be made on an receiver object of type `SummaryHTMLRenderer`. Thus, we now have an even longer $SC_{\text{line 25}}$: line 20 $\rightarrow$ line 11 $\rightarrow$ line 25.

The allocation site at line 3 is the object-sensitive context for the target method `createRenderer()`, which contains line 20, invoked at line 4. No further extension is possible, since `main()` has been reached. Now, we have $SC_{\text{line 25}}$: line 3 $\rightarrow$ line 20 $\rightarrow$ line 11 $\rightarrow$ line 25.

If we apply SCDFA to this four-point $SC_{\text{line 25}}$, $\mathcal{T}(SC_{\text{line 25}})$ will consist of all the lines except lines 16, 18 and 22. However, including line 3 does not help as it dominates line 20 in G_{ICFG}. By removing it, we obtain $ESC_{\text{line 25}}$: line 20$\rightarrow$line 11$\rightarrow$line 25 as desired. Applying SCDFA to $ESC_{\text{line 25}}$ yields the same tailored program obtained for the three-point $SC_{\text{line 25}}$.

When lengthening *SCs*, we aim to reduce infeasible paths by exploiting branch correlations. However, with longer *SCs*, SCDFA will end up using more data-flow facts, as seen in Figure 5, making it less efficient than before. So a precision/scalability tradeoff has to be made. Our key observation is to keep a *SC* extension point found if it is inside one branch in a multi-way branching statement or a target method invoked at a polymorphic call site (Figure 3), provided that the point is not in a cycle. This way, the other branches (or target methods) are infeasible (with respect to a given *SC*) and will not show up in the tailored program.

Let us return to the program understanding task at hand. From the tailored program $\mathcal{T}(ESC_{\text{line 25}})$ obtained for $ESC_{\text{line 25}}$, we can see clearly that the allocation site at line 20 in method `createRenderer()` is responsible for the rendering-related write at line 25.

4 Formalism

We first formalize SCDFA and SCEXT and then prove some properties about Tailor.

4.1 SC-based Data-Flow Analysis

We provide a context-insensitive formalization of SCDFA as context sensitivity is achieved on top of this formalization via CFL-reachability, as shown in Figure 6. Essentially, we describe the data-flow equations needed for solving its *BU* and *TD* passes in the IFDS framework [38].

The IFDS problem framework is precise and efficient for solving data-flow problems with three properties. First, the set $\mathcal{D}$ of data-flow facts is finite. Second, the domain and range of each flow, i.e., transfer function is the power set of $\mathcal{D}$, denoted $2^{\mathcal{D}}$, the lattice ordering relation $\sqsubseteq$ on $2^{\mathcal{D}}$ is $\subseteq$ or $\supseteq$, and the meet operator $\sqcap$ is $\cap$ or $\cup$. Finally, each fow function f must be distributive over $2^{\mathcal{D}}$: $\forall S_1, S_2 \in 2^{\mathcal{D}}$: $f(S_1 \sqcap S_2) = f(S_1) \sqcap f(S_2)$.

Below we describe our *BU* and *TD* passes, including their finite domain $\mathcal{D}$ and data-flow equations used. In both cases, the ordering relation $\sqsubseteq$ is $\supseteq$ and the meet operator $\sqcap$ is $\cup$. All our transfer functions given below are easily seen to be distributive over $2^{\mathcal{D}}$.

Domain. A SC_P is a set of statement sequences ending at the same statement P, with all statements identified by their line numbers (or labels). The domain $\mathcal{D}$ for BU and TD is:

$$\mathcal{D} = \bigcup_{S \in SC_P} \mathit{Suffix}(S) \tag{1}$$

where $\mathit{Suffix}(S)$ returns the set of all suffixes of S, including ϵ (the empty string). Note that ϵ is necessary when P appears in a control-flow cycle (i.e., a loop or recursion). In the IFDS framework, the data-flow facts used are generated on the fly for efficiency reasons.

We make use of the *car*, *cdr* and *cons* functions operating on sequences in the standard manner. If s and s' are two sequences, then $s +\!\!+ s'$ returns the concatenation of s and s'.

Graph Representation. The BU and TD passes operate on the G_{ICFG} representation of a program. Without loss of generality, we assume that a node, i.e., basic block in G_{ICFG} contains one single statement. Thus, a node and the statement represented by it are used interchangeably. Let ENTRY be the entry node of G_{ICFG} and EXIT be the node for P in SC_P. Given a node n, $\mathsf{succ}(n)$ ($\mathsf{pred}(n)$) is the set of its successors (predecessors) in G_{ICFG}.

***BU*: Bottom-Up Analysis.** BU computes a global property, PANTI, for every node n in G_{ICFG}, backwards against the control flow, starting at EXIT. $\mathsf{PANTI}^{\text{out}}(n)$ represents the set of suffixes in $\mathcal{D}$ that are partially anticipable at the entry of n, i.e., that appear on some outgoing path of n ending at EXIT. Thus, we have the following initialization at EXIT:

$$\mathsf{PANTI}^{\text{out}}(\mathsf{EXIT}) = \{P\} \tag{2}$$

which means that our point of interest P in SC_P is partially anticipable at the entry of EXIT.

The transfer equations at a node n in G_{ICFG} are given by:

$$\mathsf{PANTI}^{\text{in}}(n) = \bigcup_{n' \in \mathsf{succ}(n)} \mathsf{PANTI}^{\text{out}}(n') \tag{3}$$

$$\mathsf{PANTI}^{\text{out}}(n) = \mathsf{GEN}_{BU}(n) \cup \mathsf{PANTI}^{\text{in}}(n) \tag{4}$$

Due to the presence of $\mathsf{PANTI}^{\text{in}}(n)$ in (4), whatever is anticipable at the exit of n are also anticipable at the entry of n. No old data-flow facts (i.e., known suffixes) are killed, because some sequences in SC_P may share some common suffixes but have different prefixes. GEN_{BU} is applied to generate all the new suffixes partially anticipable at a node:

$$\mathsf{GEN}_{BU}(n) = \bigcup_{s \,\in\, \mathsf{PANTI}^{\text{in}}(n)} \mathsf{ADD\text{-}SUFFIX\text{-}SEEN}_{BU}(n, s) \tag{5}$$

Given a partially anticipable suffix s at the exit of a node n, $\mathit{cons}(n, s)$ will also be partially anticipable at the entry of n if $\mathit{cons}(n, s) \in \mathcal{D}$. Hence, we have:

$$\mathsf{ADD\text{-}SUFFIX\text{-}SEEN}_{BU}(n, s) = \begin{cases} \{\mathit{cons}(n, s)\} & \textbf{if } \mathit{cons}(n, s) \in \mathcal{D} \\ \varnothing & \textbf{otherwise} \end{cases} \tag{6}$$

The following lemma follows immediately from the data-flow equations (2) – (6).

▶ **Lemma 1.** *For any node n, $\epsilon \notin \mathsf{PANTI}^{\text{in}}(n)$ and $\epsilon \notin \mathsf{PANTI}^{\text{out}}(n)$ always hold.*

▶ **Example 2.** Figure 5 illustrates the BU pass with the data-flow results shown. We start with $\mathsf{PANTI}^{\text{out}}(\mathsf{EXIT}) = \mathsf{PANTI}^{\text{out}}(12) = \{w\}$ and finish with $\mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY}) = \{ncw, cw, w\}$.

***TD*: Top-Down Analysis.** *TD* computes a global property, PAVAIL, for every node n in G_{ICFG}, forwards along the control flow, starting from ENTRY and visiting only the nodes reachable in BU. The basic idea is simple. A statement sequence $S \in SC_P$ starts at ENTRY and flows forwards along the control flow and ends up with its first statement $car(S)$ removed on encountering the node representing $car(S)$. Therefore, a node n is included in the tailored program $\mathcal{T}(SC_P)$ if the remaining suffix of S that flows to the entry of a node n appears also in $\text{PANTI}^{\text{out}}(n)$ or the entire sequence S has been removed upon reaching n (when P in SC_P appears in a control-flow cycle). Formally, $\text{PAVAIL}^{\text{in}}(n)$ computes the set of suffixes $s \in \mathcal{D}$ to represent implicitly (for efficiency reasons) the fact that their corresponding prefixes p, such that $p +\!\!+ s = S$ for some $S \in SC_P$, are partially available at the entry of n.

As all the sequences in $SC_P \setminus \text{PANTI}^{\text{out}}(\text{ENTRY})$ have been filtered out by BU, we only need to focus on the ones partially anticipable at ENTRY. Hence, our initialization is:

$$\text{PAVAIL}^{\text{in}}(\text{ENTRY}) = SC_P \cap \text{PANTI}^{\text{out}}(\text{ENTRY}) \tag{7}$$

The transfer equations at a non-ENTRY node in G_{ICFG} are given by:

$$\text{PAVAIL}^{\text{in}}(n) = \bigcup_{n' \in \text{pred}(n)} \text{PAVAIL}^{\text{out}}(n') \tag{8}$$

$$\text{PAVAIL}^{\text{out}}(n) = \begin{cases} \text{GEN}_{TD}(n) & \textbf{if } \text{PANTI}^{\text{out}}(n) \neq \varnothing \\ \varnothing & \textbf{otherwise} \end{cases} \tag{9}$$

During the top-down pass, we only need to visit a node n if n is reachable during the bottom-up pass, which happens when $\text{PANTI}^{\text{out}}(n) \neq \varnothing$.

Unlike GEN_{BU}, GEN_{TD} may preserve/kill an old data-fact and generate some new ones:

$$\text{GEN}_{TD}(n) = \bigcup_{s \in \text{PAVAIL}^{\text{in}}(n)} \text{ADD-SUFFIX-EXPECTED-TO-SEE}_{TD}(n, s) \tag{10}$$

As a suffix $s \in \text{PAVAIL}^{\text{in}}(n)$ represents the fact that its corresponding prefix p, such that $p +\!\!+ s = S$ for some $S \in SC_P$, is partially available at the entry of n, we have:

$$\text{ADD-SUFFIX-EXPECTED-TO-SEE}_{TD}(n, s) = \begin{cases} \{cdr(s)\} & \textbf{if } car(s) = n \\ \{s\} & \textbf{otherwise} \end{cases} \tag{11}$$

There are two cases. If $car(s) = n$, then $cdr(s)$ is generated, indicating that $p +\!\!+ n$ is partially available at the exit of n. At the same time, s is killed (for efficiency not correctness), as s would be redundantly propagated otherwise. If $car(s) \neq n$, then s is simply preserved.

▶ **Example 3.** Figure 5 illustrates the *TD* pass with the data-flow results shown. We start with $\text{PAVAIL}^{\text{in}}(\text{ENTRY}) = SC_w \cap \text{PANTI}^{\text{out}}(\text{ENTRY}) = \{ncw\}$ and finish with $\text{PAVAIL}^{\text{in}}(\text{EXIT}) = \text{PAVAIL}^{\text{in}}(12) = \{ncw, w\}$. At node 27, $\text{PAVAIL}^{\text{out}}(27) = \{w\}$, since $\text{PAVAIL}^{\text{in}}(27) = \{cw\}$.

Tailored Program. Finally, a node n is included in $\mathcal{T}(SC_P)$ if TAILORED(n) holds:

$$\text{TAILORED}(n) = \text{PAVAIL}^{\text{in}}(n) \cap \text{PANTI}^{\text{out}}(n) \neq \varnothing \vee \epsilon \in \text{PAVAIL}^{\text{in}}(n) \tag{12}$$

The first disjunct suffices if P in SC_P is not in a cycle (in G_{ICFG}). If $s \in \text{PAVAIL}^{\text{in}}(n) \cap \text{PANTI}^{\text{out}}(n)$, then a prefix p, such that $p +\!\!+ s = S$ for some $S \in \text{PAVAIL}^{\text{in}}(\text{ENTRY})$, is partially available at the entry of n, then n must be in $\mathcal{T}(SC_P)$, since it lies on a path passing through all statements in S. If P is in a cycle, then the whole sequence S that starts at ENTRY ends up being removed eventually, resulting in $\epsilon \in \text{PAVAIL}^{\text{in}}(n)$. Then n should be included in $\mathcal{T}(SC_P)$ as well. Note that $\epsilon \notin \text{PANTI}^{\text{out}}(n)$ by Lemma 1.

▶ **Example 4.** According to the data-flow facts shown for the program given in Figure 5, the tailored program consists of all the lines except lines 38 – 40 according to (12).

4.2 SC Extension

We make use of the points-to information provided by a pointer analysis to extend a SC to reduce infeasible paths that would otherwise be introduced by SCDFA. Given a statement sequence $S \in SC$, SCEXT will lengthen it recursively by prepending the object allocation sites representing the calling contexts for the method containing its first statement, as demonstrated in Section 3.2. The general algorithm for lengthening a sequence $S : L_1 \rightarrow \cdots \rightarrow L_n$ is as follows. Suppose that L_1 resides in a method m invoked at a virtual call site. Let $A_1, \cdots, A_r$ be its all r allocation sites for creating the receiver objects on which m is invoked. Then S grows into $A_1 \rightarrow L_1 \rightarrow \cdots \rightarrow L_n, \cdots, A_r \rightarrow L_1 \rightarrow \cdots \rightarrow L_n$, and the same process continues.

With longer SCs, SCDFA will be less efficient due to more data-flow facts introduced. We will keep a SC extension point if it is *embedded* in a branch and ignore it otherwise. This way, SCEXT can enable SCDFA to avoid infeasible paths more effectively by exploiting branch correlations. A statement is said to be embedded in a branch if it appears intraprocedurally (directly) or interprocedurally (indirectly) inside a multi-way branching statement or a target method invoked at a *polymorphic call site* (i.e., a virtual call site with at least two target methods). Let *Stmts* be the set of all statements in G_{ICFG}. We use the following function $\mathit{InBranchOrVC} : \mathit{Stmts} \rightarrow \textbf{boolean}$ to capture formally this branch-embedding relation:

$$\mathit{InBranchOrVC}(s) = \begin{cases} \textbf{true} & \textbf{if } \mathit{InIntraBranch}(s) \\ \textbf{false} & \textbf{else if } \mathit{InMain}(s) \\ \mathit{InInterBranchOrVC}(s) & \textbf{otherwise} \end{cases} \tag{13}$$

where $\mathit{InIntraBranch}(s)$ determines if s appears directly in a branch or not and $\mathit{InMain}(s)$ tells us whether s appears directly in `main()` or not. $\mathit{InInterBranchOrVC} : \mathit{Stmts} \rightarrow \textbf{boolean}$ checks to see whether s is embedded in a branch interprocedurally or not:

$$\mathit{InInterBranchOrVC}(s) = \bigvee_{c \in \mathit{Caller}(m)} \Big(\mathit{InBranchOrVC}(c) \ \vee \ |\mathit{Callee}(c)| > 1 \Big) \quad \textbf{where } m \textbf{ is the method containing } s \tag{14}$$

where $\mathit{Caller}(m)$ returns the set of call sites at which m is invoked. For each call site c, there are two disjuncts. One represents a recursive application of $\mathit{InBranchOrVC}$ defined in (13) to c. The other one, $|\mathit{Callee}(c)| > 1$, evaluates to **true** if the call site c is polymorphic.

▶ **Example 5.** For the program in Figure 7, as discussed in Section 3.2, SCEXT starts with $SC_{\text{line 25}}$: line 25, grows it to $SC_{\text{line 25}}$: line 3 → line 20 → line 11 → line 25, and finally settles with $ESC_{\text{line 25}}$: line 20 → line 11 → line 25. Let us see why a SC extension point is kept or ignored. Line 3 should be ignored since $\mathit{InBranchOrVC}(\text{line 3}) = \neg \mathit{InMain}(\text{line 3}) =$ **false**. Line 20 is retained since $\mathit{InInterBranchOrVC}(\text{line 20}) = \mathit{InIntraBranch}(\text{line 20}) =$ **true**. Finally, line 11 is also retained because we have $\mathit{InInterBranchOrVC}(\text{line 11}) = \mathit{InInterBranchOrVC}(\text{line 8}) = \mathit{InInterBranchOrVC}(\text{line 5}) = |\mathit{Callee}(\text{line 5})| > 1 =$ **true**, where the call site at line 5 is polymorphic according to the points-to information provided.

In practice, SCDFA does not usually benefit from a SC extension point if it appears in a control-flow cycle (a loop or a recursion cycle) in G_{ICFG}. Such cycle-related points are identified and ignored as well. To detect recursion cycles effectively, we apply Tarjan's algorithm [48] to find strongly connected components on the call graph of the program. To detect (natural) loops, we resort to a textbook loop detection algorithm. The statements reachable directly or indirected from a loop are also considered as being part of the loop.

statement	$s, s_1, s_2, ..., s_m \in \mathbb{S}$
execution	$e : s_1 s_2 ... s_m \in \mathbb{E}^G$
sequence	$sq : s_1 s_2 ... s_m \in SC$
relation $\mathbb{E}^G \times SC$	$e \rightsquigarrow sq$
relation $\mathbb{E}^G \times \mathbb{S}$	$e \rightarrow s$
execution set	$\mathbb{E}^{sq} = \{e \in \mathbb{E}^G \mid e \rightsquigarrow sq\}$
statement set	$\mathbb{S}^{sq} = \{s \in \mathbb{S} \mid e \in \mathbb{E}^{sq}, e \rightarrow s\}$

Figure 8 Notations used in proofs.

4.3 Properties

We prove that TAILOR is sound (Theorem 3), by showing that SCDFA and SCEXT are sound with respect to G_{ICFG} (Theorems 1 and 2), and consequently, that every tailored program is *SC-executable*, i.e., executable along all execution paths passing through a given *SC*.

We make use of the notations given in Figure 8 in our proofs. $\mathbb{S}$ is a set of statements in G_{ICFG}. $\mathbb{E}^G$ represents all runtime executions of the program represented by G_{ICFG} and each execution e consists of a sequence of statements in $\mathbb{S}$. We write $e \rightsquigarrow sq$ if execution e contains all the statements in a statement sequence $sq = s_1 s_2 \cdots s_m$ in exactly the same order, ending at s_m. We write $e \rightarrow s$ if execution e contains statement s. $\mathbb{E}^{sq}$ is the set of all executions that pass through a given sequence sq. Finally, $\mathbb{S}^{sq}$ is the set of all statements that appear in all possible executions e (passing through sq), where $e \in \mathbb{E}^{sq}$.

The following theorem states that SCDFA is sound with respect to G_{ICFG}.

▶ **Theorem 1** (Soundness of SCDFA). *Let G_{ICFG} be the ICFG of a program. Let SC be given (as defined in Section 3). If $sq \in SC$, then $s \in \mathbb{S}^{sq} \Longrightarrow \mathsf{TAILORED}(s)$.*

Proof. We show that for every execution e such that $e \rightsquigarrow sq$, where $sq \in SC$, $\mathsf{TAILORED}(s)$, which is given in (12), holds for all statements s such that $e \rightarrow s$. Let $sq = s_1 s_2 ... s_m$. Since $e \rightsquigarrow sq$, every statement s_i must appear at least once in e or more in the presence of control-flow cycles. For convenience, let s_e^0 be a fictitious statement at the beginning of e. Let s_e^{m+1} be the last occurrence of s_m in e, i.e., the last statement in e. Let s_e^i be the first occurrence of s_i in e after s_e^{i-1}, where $1 \leqslant i \leqslant m$. We can now divide e into $m+1$ sub-executions, $e_1, e_2, \cdots, e_{m+1}$, where e_i, represents the sub-sequence between s_e^{i-1} (exclusive) and s_e^i (inclusive). We will still write $e_i \rightarrow s$ if e_i contains a statement s.

As $e \rightsquigarrow sq$ is an execution, then e represents a realizable path, which must appear in G_{ICFG}. In SCDFA, its *BU* and *TD* passes are distributive over $2^{\mathcal{D}}$ given in (1), performed context-sensitively. Thus, we only need to focus on this path. Note that in our formulation of SCDFA, a statement s_i and its corresponding node n_i in G_{ICFG} are used interchangebly.

During the *BU* pass in terms of (2) – (6), we must have:

$$\forall\ 1 \leqslant i \leqslant m : \forall\ n_i\ s.t.\ e_i \rightarrow n_i : s_i s_{i+1} \cdots s_m \in \mathsf{PANTI}^{\text{out}}(n_i) \tag{15}$$

$$\forall\ n_{m+1}\ s.t.\ e_{m+1} \rightarrow n_{m+1} : s_m \in \mathsf{PANTI}^{\text{out}}(n_{m+1}) \tag{16}$$

which implies $sq \in \mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY})$ (since $\forall\ n_1\ s.t.\ e_1 \rightarrow n_1 : s_1 s_2 \cdots s_n \in \mathsf{PANTI}^{\text{out}}(n_1)$).

During the *TD* pass, $sq \in \mathsf{PAVAIL}^{\text{in}}(\mathsf{ENTRY})$ by (7). By (8) – (11), we must have:

$$\forall\ 1 \leqslant i \leqslant m : \forall\ n_i\ s.t.\ e_i \rightarrow n_i : s_i s_{i+1} \cdots s_m \in \mathsf{PAVAIL}^{\text{in}}(n_i) \tag{17}$$

$$\forall\ n_{m+1}\ s.t.\ e_{m+1} \rightarrow n_{m+1} : \epsilon \in \mathsf{PAVAIL}^{\text{in}}(n_{m+1}) \tag{18}$$

(Note that (18) is needed only if e_{m+1} is non-empty, which happens when s_m is in a control-flow cycle.) Hence, $\mathsf{TAILORED}(s)$ holds for every statement s such that $e \rightarrow s$. ◀

For a SC, we write $\alpha(G_{\text{ICFG}}, SC)$ to represent the set of all executions in G_{ICFG} that pass at least one sequence in SC, i.e., $\alpha(G_{\text{ICFG}}, SC) = \bigcup_{sq \in SC} \mathbb{E}^{sq}$. Theorem 1 can also be stated equivalently as follows: TAILORED(s) holds for every $s \in \{s \mid e \in \alpha(G_{\text{ICFG}}, SC), e \to s\}$.

The following theorem states that SCEXT is sound since only infeasible paths are ignored.

▶ **Theorem 2** (Soundness of SCEXT). *Let ESC be extended from a given SC by applying* SCEXT *in G_{ICFG}, then $\alpha(G_{\text{ICFG}}, SC) = \alpha(G_{\text{ICFG}}, ESC)$.*

Proof. According to the algorithm of SCEXT operating in G_{ICFG} (Section 4.2), all possible object-sensitive calling contexts for a method containing the first point F in a sequence of SC are considered as the extension points of F. According to (13) and (14), no feasible paths with respect to SC are excluded if an extension point of F is not selected. Thus, only infeasible paths with respect to SC are ignored when SC is extended into ESC this way. ◀

▶ **Theorem 3** (Soundness of TAILOR). TAILOR *is sound with respect to G_{ICFG}.*

Proof. Follows from Theorems 1 and 2. ◀

▶ **Theorem 4** (SC-Executability). *$\mathcal{T}(SC)$ obtained in G_{ICFG} is SC-executable.*

Proof. By Theorem 3, $\alpha(G_{\text{ICFG}}, SC)$ is included in the tailored program. ◀

5 Implementation

We have implemented TAILOR (`http://www.cse.unsw.edu.au/~corg/tailor`) in SOOT, a framework for analyzing and optimizing Java programs [51]. To build the ICFG for a program, we apply SOOT's SPARK pointer analysis [27]. During the ICFG construction, SOOT models the effects of native methods by using abstract Java code and creates the corresponding control-flow edges. In addition, SOOT considers both explicit and implicit exceptions and treats the exceptional edges as normal control-flow edges. Finally, SOOT models thread creation and running as method calls by assuming that threads execute in a sequential order. How to build ICFGs to support multi-threading soundly, precisely and scalably for Java programs is a big challenge in its own right.

TAILOR has two main components, SCEXT and SCDFA. When extending *SCs*, we make use of the points-to information provided by SPARK to find the required object allocation sites object-sensitively. To perform SCDFA, we choose HEROS [10] as the IFDS solver for its *BU* and *TD* data-flow problems, because it can be easily plugged into the SOOT framework.

6 Evaluation

Program tailoring is designed to be sound for a program (with respect to its ICFG, as proved in Section 4), with useful precision and good scalability for large object-oriented programs. This section serves to evaluate the last two goals by answering three research questions:
RQ1: Is TAILOR useful to support program debugging and understanding, in practice?
RQ2: Is TAILOR useful to support program analysis, in practice?
RQ3: Is TAILOR scalable for large object-oriented programs, in practice?

To address RQ1 and RQ2, we conduct two real-world case studies. In one study, we demonstrate that TAILOR can assist a state-of-the-art slicing tool, a thin slicer [47] implemented in WALA [52], to simplify debugging and understanding tasks. In the other study, we demonstrate that TAILOR can enable a sophisticated pointer analysis algorithm, S-2OBJ [24],

Table 1 Program characteristics. For each program, the numbers are produced for both the application and library code, including only reachable classes, methods and statements by Spark [27].

Application	Description	#Classes	#Methods	#ByteCodes	LOC
ANTLR (2.7.2)	a recognizer and parser generator	2049	13,751	261,727	90,404
Avrora (1.7.117)	an assembly program simulator	3196	17,186	276,340	92,505
Eclipse (4.5)	IDE	2517	16,953	305,575	106,640
Apache™ FOP (0.20.5)	a formatting-objects processor	4681	28,105	492,686	171,087
JBoss AS (4.0.2)	an application server	4039	25,634	448,163	154,290
PMD (4.0)	a source code analyzer	4234	26,623	467,249	161,300
Apache Tomcat™ (8.0.24)	a Java Servlet container	3920	25,157	432,652	150,074

provided in a state-of-the-art pointer analysis tool for Java, Doop [14], to investigate the multi-object typestate (reflective) behavior in programs for which S-2Obj is unscalable as a whole-program analysis. In both studies, all *SCs* used are deduced from the results generated by state-of-the-art clients, Clara [9] and Solar [29], rather than injected manually.

To address RQ3, we perform a stress test on Tailor by using a large number of randomly generated *SCs*. Tailor scales well to relatively large Java programs, suggesting that program tailoring represents an attractive option as a practical tool.

Our experiments are carried out on an Xeon E5-2650 2GHz machine with 64GB RAM. We have selected seven large and diverse real-world Java programs under a large library, JDK 1.6.0_45, described in Table 1. For each program, with its bytecode representation generated by Soot [51], all the statistics are calculated by using Soot's Spark pointer analysis [27].

6.1 RQ1: Program Debugging and Understanding

Wala [52] includes a traditional slicer [21] and a thin slicer [47] (denoted TSlicer), with industry-strength implementations. Traditional slicing does not scale to large object-oriented programs due to the key bottleneck in handling of the heap [47]. Thin slicing [47] alleviates the bottleneck by including only producer statements that affect directly the values at a program point. This unsound design can facilitate program debugging and understanding [25, 50, 59].

By focusing on producer statements, TSlicer is surprisingly effective, by producing often small (or thin) slices quickly. In this study, we show that Tailor can make TSlicer even more effective, as highlighted in Figure 1, by exploiting the temporal order of statements in *SCs* to prune away more statements irrelevant to *SCs*. In addition, Tailor achieves this improved precision by trimming a program more efficiently than TSlicer does. These results are significant given TSlicer's unsoundness (even for G_{ICFG}) and well-tuned implementation in Wala, demonstrating clearly the practical benefits of our *SC*-oriented program tailoring.

To ensure a fair comparison between Tailor (implemented in Soot) and TSlicer (implemented in Wala), we have configured Soot to minimize the differences in the ICFGs constructed for a program in both frameworks. There are four main contributing factors: (1) pointer analysis, (2) reflection analysis, (3) exception handling, and (4) native code handling. Our bottom-line is to make sure that Tailor is never more precise than TSlicer in dealing with (1) – (4). For (1), we select Wala's *VanillaZeroOneCFA* option to perform its allocation-site-sensitive pointer analysis, which is more precise than Soot's Spark pointer analysis, as Spark merges some `java.lang.String` allocation sites for improving performance. For (2), we use Wala's *no_flow_to_casts* option to resolve reflective calls. In Soot, we have taken advantage of Solar [29] to perform reflection resolution in the same way. For (3) and (4), Wala and Soot model the same native methods in JDK 1.3 and handle both explicit and implicit exceptions with some differences. However, these differences do not affect the

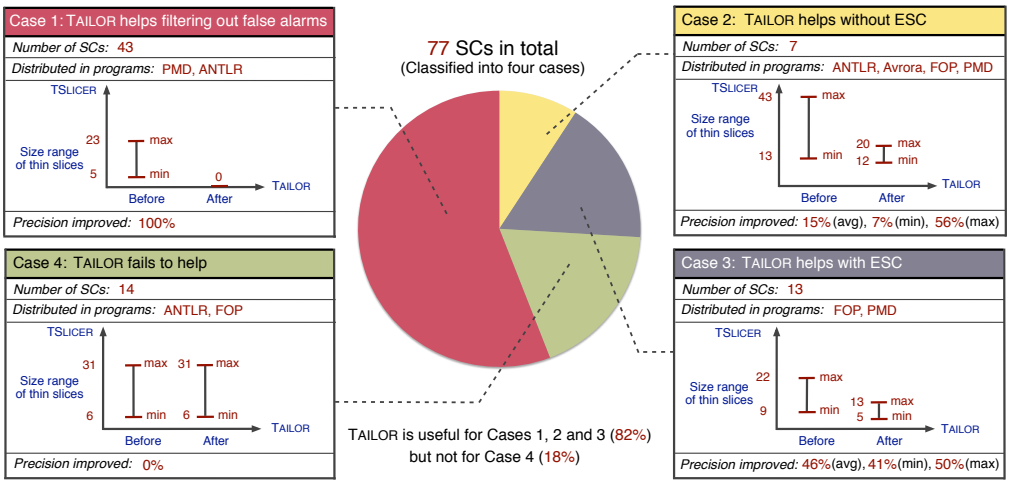


Figure 9 A case study demonstrating how TAILOR enables TSLICER to remove more *SC*-irrelevant statements based on the *SCs* deduced from the results reported by a typestate analysis, CLARA [9].

precision achievements obtained, validated by having inspected all results manually.

Both TAILOR and TSLICER trim a program based on the writer/OutputStream-related *SCs* deduced from the results reported by a state-of-the-art typestate analysis tool, CLARA [9]. TSLICER is run context-sensitively by using the last point in each *SC* as its slicing criterion with the variables of interest selected automatically. However, the results in the following cases are excluded: (1) CLARA crashes due to runtime exceptions in the case of `Eclipse`, `JBoss` and `Tomcat`, (2) TSLICER is unscalable for a criterion (within 1 hour), and (3) the size of a thin slice is less than 5 (small enough for human consumption). Finally, 77 *SCs* are considered in total, involving 66 errors and 11 program points (like the one in Figure 7) for our debugging and understanding tasks. All these *SCs* are provided in our artifact.

Results and Analysis Figure 9 presents the final results, with all the 77 *SCs* classified into four cases. In each case, we give the number of *SCs* included, the names of *SC*-contributing programs, the size ranges of TSLICER's thin slices before and after the *SC*-irrelevant statements detected by TAILOR are removed, as well as the minimum, maximum and average precision improvements achieved. Note that WALA's traditional slicer is unscalable for any *SC* (within 1 hour). Below we first analyze each case and then make some remarks.

Case 1. There are 43 *SCs* distributed in two programs, `ANTLR` and `PMD`. TSLICER has produced thin slices ranging from 5 to 23 statements, requiring further human analysis efforts. In contrast, TAILOR has produced only zero-sized tailored programs, declaring all the 43 errors as false alarms with respect to the given *SCs*. Furthermore, as CLARA is sound (with respect to G_{ICFG}), all the 43 errors are false alarms for the 43 reported locations.

Let us consider a *SC* from `PMD`. CLARA reports a "write after close" typestate error for a call to `writer.write()` at line 33 in class `net.sourceforge.pmd.renderers.TextRenderer`, together with four two-statement sequences of method calls leading to this potential error location: line 290 → line 325, line 290 → line 337, line 292 → line 325, and line 292 → line 337. These five calls are distributed in classes `net.sourceforge.pmd.PMD` and the afore-mentioned class `TextRenderer`. Given this four-sequence *SC* ending at line 33, TAILOR recognizes that all these sequences are infeasible since line 33 is not reachable from lines 325 and 337.

Clara reports such false errors as it is partially context-sensitive and intraprocedurally but not interprocedurally flow-sensitive. This is a typical trade-off made by static analyses, which must, for example, reason about complicated typestate or protocol information as well. Otherwise, full context- and flow-sensitivity is unattainable scalably for large programs [9]. Unlike these client analyses, Tailor reasons about the (un)reachability of a statement towards a *SC*, making it substantially more amenable to a fully context- and flow-sensitive analysis. Tailor's success in this case is potentially replicable for other analysis tools [9, 29, 37].

Case 2. There are 7 *SCs* spread across `ANTLR`, `Avrora`, `FOP` and `PMD`. In this case, SCEXT is not useful. These *SCs* share the same characteristic as $SC_{\text{line 12}}$: line 2→line 27→line 12 from our motivating example in Figure 2 (Section 2). For each *SC*, Tailor has succeeded in removing some *SC*-irrelevant statements in some branches from TSlicer's thin slices.

Case 3. There are 13 *SCs* found in `FOP` and `PMD`. In this case, Tailor will be ineffective unless SCEXT is turned on. We take a *SC* in `FOP` ending at line 101 in class `CommandLineStarter` to show how Tailor can simplify debugging tasks for object-oriented programs enormously. Given this point of interest, TSlicer returns a thin slice containing nine statements, which are distributed in five classes, including `AWTStarter`, `PrintStarter` and `CommandLineStarter` in package `org.apache.fop.apps`, where the first two are the subclasses of the last one. The first statement of this *SC* resides in `CommandLineStarter`'s `run()` method, which is overridden in the two subclasses. During the SC extension, the object allocation site at line 522 in the `CommandLineOptions` class is found to be the sole object-sensitive context for `CommandLineStarter`'s `run()` method. Therefore, with this extension point, Tailor is able to remove four irrelevant statements, line 94 in `AWTStarter`, line 91 in `PrintStarter`, and lines 494 and 508 in `CommandLineOptions`, from TSlicer's thin slice, saving a human a lot of debugging effort on navigating through many irrelevant classes unnecessarily.

Case 4. There are 14 *SCs* found in `ANTLR` and `FOP`. Tailor fails to reduce TSlicer's thin slices any further. By including producer statements (and ignoring the others unsoundly), TSlicer has happened to eliminate all the *SC*- irrelevant statements removed by Tailor.

Remarks. First, Tailor has succeeded in making 82% (56%) of TSlicer's 77 thin slices smaller (empty), as shown in Figure 9, even though TSlicer is known to return small slices unsoundly. Second, Tailor aims to eliminate *SC*-irrelevant statements. As discussed in Section 2 and elaborated in Case 3, removing several or even just one *SC*-irrelevant statement can save a lot of debugging effort, particularly for large object-oriented programs. Finally, Tailor is fast, as compared with TSlicer in Figure 10. To make the analysis times for Tailor visible, the longest analysis times spent by TSlicer on several *SCs* are depicted at the top-left corner. In Case 4, Tailor is ineffective, but no harm is done, as Tailor is fast. Without Tailor, the practical benefits reaped from exploiting the temporal order in the other *SCs* in Cases 1 – 3 will be missed.

6.2 RQ2: Program Analysis

In this second case study, we demonstrate that Tailor can be invaluable for pointer analysis (the foundation for virtually all other analyses). In particular, we show how Tailor can enable today's most sophisticated pointer analysis algorithms, which are unscalable for a program, to perform a more focused and thus potentially scalable analysis to its specific

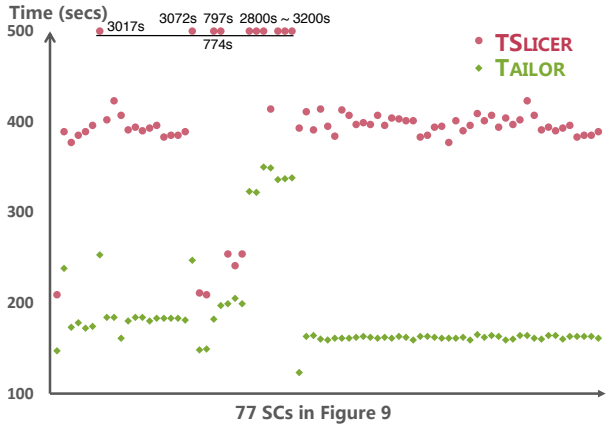


Figure 10 Efficiency of TAILOR vs. TSLICER.

parts that contain usually hard-to-analyze language features such as reflection [30]. For a Java program, a pointer analysis requires a reflection analysis to resolve part of its call graph representing reflective calls, and conversely, a reflection analysis requires the points-to information from a pointer analysis to discover the reflective targets at a reflective call site.

Reflection analysis finds many real-world applications, such as bug detection and security analysis [5, 11, 28, 31], in its own right. Despite recent advances [28, 29, 45], a sophisticated reflection analysis does not co-exist well with a sophisticated pointer analysis, since the latter is unscalable for large programs [28, 29, 31, 45]. If a scalable but imprecise points-to analysis is used instead, the reflection analysis may introduce many false call graph edges [29, 45], making its underlying client applications to be too imprecise to be practically useful.

We show how TAILOR can alleviate this problem. We choose DOOP [14], a state-of-the-art pointer analysis framework for Java, and focus on its three pointer analyses, 1-CALL, S-2TYPE and S-2OBJ. 1-CALL is usually the most scalable but most imprecise. S-2OBJ is the most precise but the most unscalable. S-2TYPE enjoys both precision and scalability, with its precision being always not better than S-2OBJ [24]. For reflection analysis, we choose SOLAR [29], a state-of-the-art reflection analysis built on top of DOOP. Our program analysis task is to investigate the multi-object typestate behavior at some reflective call sites as precisely as possible, by running SOLAR with S-2OBJ. If S-2OBJ is scalable for a program, we are done. Otherwise, we run SOLAR together with 1-CALL, and if that is unscalable, with S-2TYPE on the same program. If either is scalable, we treat SOLAR as a client analysis for producing the required *SCs*. To investigate the behavior at a reflective call site P, say, `m.invoke()`, we let its SC_P be all possible sequences of API method calls ending at P, such as `Class.forName()` $\rightarrow$ `c.getMethod()` $\rightarrow$ `m.invoke()`, found by SOLAR. Then we run S-2OBJ on the tailored program $\mathcal{T}(SC_P)$. If SC_P represents all possible sequences of method invocations for P, then $\mathcal{T}(SC_P)$ exhibits the same behavior at P as in the whole program (Theorem 3). Otherwise, a SC_P-specific behavior at P is analyzed.

For the seven applications listed in Table 1, S-2OBJ is only unscalable for `Eclipse`, `Tomcat` and `JBoss`, which will therefore be the focus of our second study. Figure 11 shows the number of reflective targets resolved at all call sites to `Class.newInstance()`/`Method.invoke()` (the most commonly used [28], in practice) in the application code of `Eclipse`, `Tomcat` and `JBoss`, by 1-CALL (the red dots) and S-2TYPE (the green dots from `JBoss` as 1-CALL is unscalable).

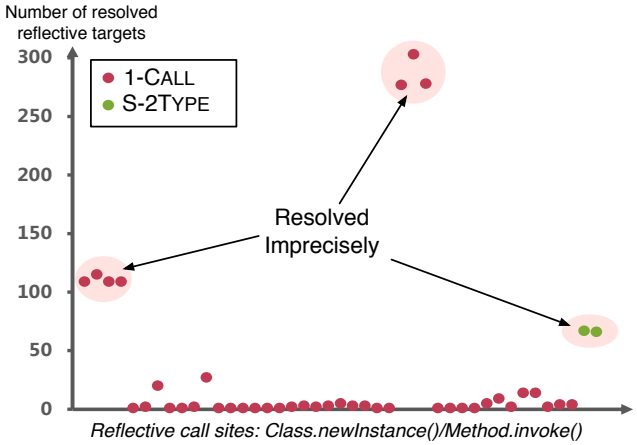


Figure 11 Reflection resolution for `Eclipse`, `Tomcat` and `JBoss`.

Table 2 A case study demonstrating how TAILOR enables a sophisticated pointer analysis to scale better by analyzing the reflective behavior at the specific parts of a program.

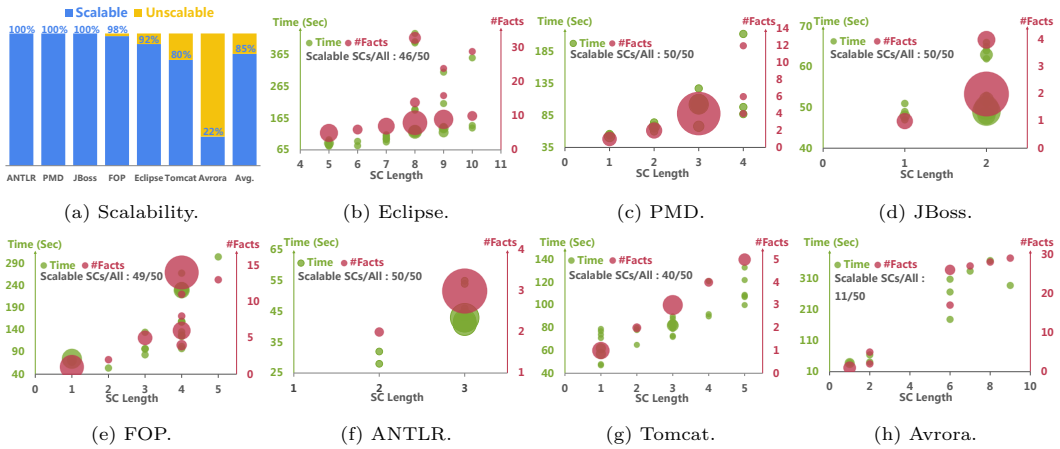
SC	Efficiency (Analysis Times)					Precision (Reflective Targets Resolved)				
						Number of Targets Resolved			Precision Improved Over	
	Before TAILOR			TAILOR	After TAILOR	Before TAILOR		After TAILOR		
	1-CALL	S-2TYPE	S-2OBJ		S-2OBJ	1-CALL	S-2TYPE	S-2OBJ	1-CALL	S-2TYPE
Eclipse-1	43m45s	129m39s	>10h	3m1s	37m25s	109	27	12	88.9%↑	55.6%↑
Eclipse-2				6m20s	>10h	109	28	—	—	—
Eclipse-3				2m29s	>10h	115	33	—	—	—
Eclipse-4				4m7s	>10h	109	28	—	—	—
Tomcat-1	42m0s	80m35s	>10h	4m11s	61m9s	303	116	1	99.7%↑	99.1%↑
Tomcat-2				3m49s	61m24s	278	1	0	100%↑	100%↑
Tomcat-3				3m42s	>10h	277	41	—	—	—
JBoss-1	>10h	171m38s	>10h	10m59s	6m14s	N/A	67	4	N/A	94%↑
JBoss-2				4m19s	6m12s	N/A	66	1	N/A	98.5%↑

There are a total of nine reflective call sites with a high number of reflective targets resolved, possibly imprecisely by 1-CALL or S-2TYPE, which result in a total of nine *SCs* deduced from their corresponding SOLAR analysis.

Table 2 shows our results. For the nine *SCs* given (in Column 1), TAILOR enables S-2OBJ to run scalably on the tailored programs obtained for five *SCs*, *Eclipse-1*, *Tomcat-1*, *Tomcat-2*, *JBoss-1* and *JBoss-2* (in 6 – 62 minutes) with significant precision improvements in terms of the number of reflective targets reduced. Without TAILOR, no existing reflection analyses [28, 29, 31, 45] can achieve such precise results automatically for their corresponding reflective calls. Note that traditional slicing [21] is unscalable given the last points in the nine *SCs* (out of (64GB) memory after 2 hours each) while thin slicing [47] (designed for program debugging and understanding only) is unsound and is thus non-applicable in this setting.

S-2OBJ remains unscalable for four *SCs*, *Eclipse-2*, *Eclipse-3*, *Eclipse-4* and *Tomcat-3*, partly because S-2OBJ is precise but slow (being basically 2-object-sensitive [24]) and partly because reflection-rich object-oriented programs such as `Eclipse` and `Tomcat` are difficult to analyze both precisely and scalably, despite recent advances [28, 29, 45]. TAILOR produces the tailored programs for the nine *SCs* in less than 11 minutes each (Column 5), making it possible for the five out of nine *SCs* to be analyze precisely than before (Columns 7 – 11).

Let us examine the *SC* denoted by *Tomcat-1*. Under 1-CALL, SOLAR identifies an imprecisely resolved `newInstance()` call (at line 268 in class `Bootstrap`) with 303 targets.



(a) Scalability. (b) Eclipse. (c) PMD. (d) JBoss.

(e) FOP. (f) ANTLR. (g) Tomcat. (h) Avrora.

Figure 12 Scalability of SCDFA on 50 randomly generated *SCs* per program (with a 10-minute budget per *SC*). "#Facts" represents the number of data-flow facts in $\mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY})$.

For this call, SOLAR reports four related `Class.forName()` and `loadClass()` calls, at lines 1232, 1265, 1299 in class `WebappClassLoaderBase` and line 265 in class `Bootstrap`, forming a four-sequence *SC* (of length 2 each). When analyzing `Tomcat` as a whole, S-2OBJ does not terminate in 10 hours. Given *Tomcat-1*, TAILOR produces a tailored program, $\mathcal{T}(\textit{Tomcat-1})$, in about 4 minutes. Given $\mathcal{T}(\textit{Tomcat-1})$, S-2OBJ finishes in about 61 minutes, enabling SOLAR to resolve the `newInstance()` call (line 268) precisely to be class `Catalina` as its target.

We have inspected manually the five *SCs* scalably analyzed and found that no true reflective targets are missed for the call sites at their last points. For *Tomcat-2*, Column 9 has a 0, because its last point, the `newInstance()` call (at line 595 in class `RewriteValue`), reported by 1-CALL is not reachable (from `main()`) and thus filtered out by S-2OBJ.

6.3 RQ3: Scalability

TAILOR has two main components, SCEXT and SCDFA, with the latter dominating the total analysis time. We perform a stress test to investigate how well SCDFA scales, in practice. For each of our seven applications listed in Table 1, we select randomly 50 statements as the potential points of interest, then apply SCEXT to these points, and finally, run SCDFA on the 50 extended *SCs*. For a total of seven applications, a total of 350 *SCs* are generated.

Figure 12 gives the final results. According to Figure 12a, SCDFA is scalable for 296 (85%) out of the 350 *SCs* generated (with a 10-minute budget per *SC*). Figures 12b – 12h provide more details. For each application, we give the number of scalable *SCs*, and for each scalable *SC*, its length (in the x-axis), its analysis time (in the left y-axis) and the maximum number of data-flow facts (suffixes) reaching ENTRY, i.e., $|\mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY})|$ (in the right y-axis). In the legend for each *SC*, there is a green dot representing its analysis time and a red dot representing its $|\mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY})|$. The data plotted for a fixed *SC* length can be understood as follows. First, if there are n *SCs* with the same analysis time ($|\mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY})|$), then its green (red) dot is n times as large as the green (red) dot in the legend. Second, the analysis time of a *SC* always increases as $|\mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY})|$ increases (allowing one to find its associated red dot given a green dot). This is expected as the time complexity of SCDFA is $O(ED^3)$, where D is the size of the data-flow facts used, i.e., $|\mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY})|$. In general, $|\mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY})|$ increases as the length of its underlying *SC* increases. However, this is not absolute, as the length of a *SC* is defined to be the length of its longest sequence (with the other shorter sequences ignored).

Finally, we explain why SCDFA is unscalable for many *SCs* in `Tomcat` and `Avrora` and discuss some possible solutions. For `Tomcat`, SCDFA is unscalable for 10 *SCs*, because many of their extension points introduced by SCEXT could have been avoided if a more precise pointer analysis (than Soot's Spark pointer analysis) is used.

For `Avrora`, SCDFA is unscalable for 39 *SCs*, due to a special programming pattern used in this (simulator) application. One factory class `cck.util.ClassMap` is used to create all its simulation and platform classes, e.g., `SensorSimulation`, residing in a total of 13 packages. As a result, most of the SC extension points introduced by SCEXT happen to land in this factory class. In addition, there are many object allocation sites for this class in the program, making all of them eligible as SC extension points and consequently increasing the number of data-flow facts used, i.e., $|\mathsf{PANTI}^{\mathrm{out}}(\mathsf{ENTRY})|$. One possible improvement is to make SCEXT pattern-aware to avoid some SC extensions that would otherwise be introduced.

6.4 Limitations

We observe that program tailoring scales better than program slicing, as the overall design of tailoring (with its object-sensitive conceptualization of SCEXT and its IFDS formulation of SCDFA laid out in Sections 1.1 and 1.2) is more amenable to efficient implementation with useful precision, as validated with our prototyping system, Tailor. However, there are still spaces for performance improvement. According to our experimental results presented in Section 6.3, a more intelligent SCEXT is needed to deal with programs such as `Avrora` more effectively. Applying a pattern-aware pre-analysis to recognize some unscalability-inducing SC extension points may be a viable solution worth trying in future work.

Tailor is practically useful in program debugging and understanding as well as program analysis, as demonstrated with two case studies. However, Tailor is expected to be more effective if we can avoid introducing irrelevant statements in loops. As explained in Section 4.2, SCEXT presently gives up a SC extension point inside a control-flow cycle but may miss an opportunity for avoiding infeasible paths at this point, with a tradeoff made favoring scalability over precision. One possible improvement is to leverage slicing to remove irrelevant statements that do not produce any data dependence for the statements in a *SC*. How to combine tailoring and slicing scalably is non-trivial but will be an interesting future work.

7 Related Work

In addition to the related work mentioned earlier, we review some other related research.

Program Slicing. There are dozens of slicing techniques proposed, including amorphous slicing [19], parametric slicing [15], interface slicing [6] and specification slicing [4]. However, none of these is close to program tailoring, as tailoring is the first to exploit a sequential criterion and designed to scale for today's large object-oriented programs. For past slicing techniques and their design goals, we refer to some survey articles [8, 20, 43, 49]. Below we examine the most closely related ones, by focusing on their connections with this work.

Traditional slicing [53] and thin slicing [47] have been introduced and compared in Section 2 and evaluated in Section 6. In general, most existing static slicing methods [20, 43, 49] make use of the traditional slicing criterion (defined in [53] and used in Section 2) to express their points of interest but may slice a program differently according to different goals. For example, thin slicing [47] considers only producer statements that may have direct effects at a point. Although unsound, thin slicing may exclude many distracting statements, easing significantly program debugging and understanding (which is its goal aimed for).

Program tailoring focuses on a totally different criterion, known as sequential criterion, which captures the temporal order of invocation sequences of related methods naturally inherent in a program. As explained in Section 2 and demonstrated in Section 6, such ordering information enables tailoring to improve the effectiveness of a modern thin slicing tool, and potentially other slicing tools, e.g., a recently proposed value slicer [26].

Path slicing [23] takes as input a program path (or trace) to a target point and tries to eliminate the statements that are irrelevant towards the reachability of the point. Given a SC_P lying on a path with P as a target point, path slicing may remove all points in SC_P except P as long as they do not affect the reachability, i.e., feasibility of the path to P (from `main()`). In contrast, program tailoring focuses on determining the reachability of statements towards a SC so that the temporal order specified in the sequences in SC is respected.

Program chopping [22,39] considers two sets of variables, one at a source point and one at a sink point, as its slicing criterion. It identifies the statements that transmit the effects from the source to the sink, by applying a forward slicing at the source and a backward slicing at the sink. Tailoring is different in three aspects. First, chopping focuses on two points but tailoring focuses on the temporal order specified by a sequential criterion consisting of possibly many statement sequences of arbitrary lengths (sinking at the same point). Second, chopping, which relies on traditional slicing, does not scale to large object-oriented programs. Finally, SCEXT is unique as it enables tailoring to work effectively, in practice. In fact, these differences are also what distinguish tailoring from other slicing techniques [8, 20, 43, 49].

API Protocol Analysis. We consider typestate analysis [9, 16, 34] as a special kind of API protocol analysis as the abstract states of an object are usually affected by API calls. API protocol analysis [7, 37] reports related invocation sequences of API methods, indicating a kind of semantic (or ordering) information in the program, which is ignored by program slicing. Such API invocation sequences can be specified manually or mined automatically [1, 3, 17, 36, 42, 60]. SCEXT can be seen as a new method for mining protocols in object-oriented programs automatically, but at a coarser granularity, based on the object-sensitive calling contexts of a method. We expect Tailor to become more effective when longer sequences (with richer semantic information) are considered as *SCs*.

Pointer Analysis. SCEXT exploits the concept of object-sensitivity [33] to extend *SCs* in order to avoid infeasible paths that would otherwise be introduced by SCDFA. We refer to [44] on why object-sensitivity is more effective than call-site-sensitivity in representing calling contexts when analyzing object-oriented programs context-sensitively. Over the years, many whole-program pointer analyses for Java have been developed [24, 27, 44, 54, 55]. In Section 6.2, we show how Tailor enables a precise whole-program pointer analysis [24] to scale better on a SC-relevant part of the program in order to analyze the reflection behavior at a program point affected by SC precisely. In principle, the reflective behavior at a program point can also be answered by raising points-to queries on-demand instead. In practice, however, existing demand-driven pointer analyses for Java [32, 40, 41, 46, 58] either ignore hard language features such as reflection and dynamic class loading or assume that they have been handled by a pre-analysis (e.g., when G_{ICFG} is built). In fact, how to analyze such hard language features is a big challenge in its own right [30], despite recent advances on static handling of reflection and dynamic class loading [5, 28, 29, 31, 35, 45, 56, 57]. In this setting, we are not aware of any SC-aware demand-driven pointer analysis, not to mention its scalability to large object-oriented programs flow- and context-sensitively.

8 Conclusions and Future Work

This paper brings a new dimension to program slicing by introducing a sequential criterion. The temporal ordering constraints that appear in a sequential criterion, which can be, e.g., one or several sequences of API usage calls leading to a program point, are naturally inherent in many real-world applications, but have not been exploited in program slicing before. Accordingly, we propose program tailoring, a new technique to trim a program by a sequential criterion soundly (with respect to a given ICFG) and scalably (for reasonably large object-oriented programs). Regarding this new research work — program tailoring, in theory, we have formalized it, proved its soundness (with respect to a given ICFG), and discussed its advantages and limitations. In practice, we have produced a soon-to-be released open-source implementation, Tailor, and demonstrated its usefulness for improving the effectiveness of existing slicing techniques in program debugging and understanding and for supporting program analysis with large Java programs due to its good scalability.

Program tailoring builds on a natural connection with several research fields: program slicing, API protocol (or specification) mining, and program analysis. Therefore, a lot of interesting future work is anticipated to investigate their interplay. One possibility is to combine tailoring and slicing to eliminate irrelevant statements that cannot be eliminated by either alone scalably for a given task. Another is to combine API protocol analysis with SC extension to enable the latter to exploit richer semantic information available. Finally, program tailoring also provides new opportunities for developing program analyses (e.g., pointer analysis) that are focused and partial, paying closer attention to specific parts of the program, where some hard language features need to be analyzed precisely and scalably.

Acknowledgements. The authors wish to thank the anonymous reviewers for their valuable comments.

References

1 Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. FSE '07.
2 Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle, and Westley Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. SAS '02.
3 Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. POPL '02.
4 Min Aung, Susan Horwitz, Rich Joiner, and Thomas Reps. Specialization slicing. *ACM Trans. Program. Lang. Syst.*, 2014.
5 Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. Static analysis of implicit control flow: Resolving Java reflection and android intents. ASE '15.
6 Jon Beck and David Eichmann. Program and interface slicing for reverse engineering. ICSE '93.
7 Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. ECOOP '09.
8 David Binkley and Mark Harman. A survey of empirical results on program slicing. *Advances in Computers.*, 2004.
9 Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. ICSE '10.
10 Eric Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. SOAP '12.

11 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. ICSE '11.
12 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. OOPSLA '09.
13 Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. PLDI '02.
14 DOOP. `http://doop.program-analysis.org`.
15 John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. POPL '95.
16 Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 2008.
17 Mark Gabel and Zhendong Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. FSE '08.
18 GrammaTech. GrammaTech static analysis. `http://www.grammatech.com`.
19 Mark Harman and Sebastian Danicic. Amorphous program slicing. IWPC '97.
20 Mark Harman and Rob Hierons. An overview of program slicing. *Software Focus.*, 2001.
21 Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 1990.
22 Daniel Jackson and Eugene J. Rollins. A new model of program dependences for reverse engineering. SIGSOFT '94.
23 Ranjit Jhala and Rupak Majumdar. Path slicing. PLDI '05.
24 George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. PLDI '13.
25 Andrew J. Ko and Brad A. Myers. Extracting and answering why and why not questions about Java program output. *ACM Trans. Softw. Eng. Methodol.*, 2010.
26 Shrawan Kumar, Amitabha Sanyal, and Uday P. Khedker. Value slice: A new slicing concept for scalable property checking. TACAS '15.
27 Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. CC '03.
28 Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. Self-inferencing reflection resolution for Java. ECOOP' 14.
29 Yue Li, Tian Tan, and Jingling Xue. Effective soundness-guided reflection analysis. SAS'15.
30 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *CACM*, 2015.
31 Benjamin Livshits, John Whaley, and Monica Lam. Reflection analysis for Java. APLAS'05.
32 Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. An incremental points-to analysis with CFL-reachability. CC '13.
33 Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 2005.
34 Nomair A. Naeem and Ondřej Lhoták. Typestate-like analysis of multiple interacting objects. OOPSLA '08.
35 Phung Hua Nguyen and Jingling Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. ACSC '05.
36 Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. FSE '09.
37 Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking API protocol conformance with mined multi-object specifications. ICSE '12.
38 Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. POPL '95.
39 Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. SIGSOFT '95.

40 Lei Shang, Yi Lu, and Jingling Xue. Fast and precise points-to analysis with incremental CFL-reachability summarisation. ASE '12.

41 Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. CGO '12.

42 Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. ISSTA '07.

43 Josep Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 2012.

44 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. POPL '11.

45 Yannis Smaragdakis, George Kastrinis, George Balatsouras, and Martin Bravenboer. More sound static handling of Java reflection. APLAS '15.

46 Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. PLDI '06.

47 Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. PLDI '07.

48 Robert Tarjan. Depth first search and linear graph algorithms. *SICOMP*, 1972.

49 Frank Tip. A survey of program slicing techniques. *J Program Lang*, 1995.

50 Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective taint analysis of web applications. PLDI '09.

51 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. CASCON '99.

52 WALA. T.J. Watson libraries for analysis. `http://wala.sf.net`.

53 Mark Weiser. Program slicing. ICSE '81.

54 John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. PLDI '04.

55 Xiao Xiao and Charles Zhang. Geometric encoding: Forging the high performance context sensitive points-to analysis for Java. ISSTA '11.

56 Jingling Xue and Phung Hua Nguyen. Completeness analysis for incomplete object-oriented programs. CC '05.

57 Jingling Xue, Phung Hua Nguyen, and John Potter. Interprocedural side-effect analysis for incomplete object-oriented software modules. *Journal of Systems and Software*, 2007.

58 Dacong Yan, Guoqing Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for java. ISSTA '11.

59 Sai Zhang and Michael D. Ernst. Which configuration option should I change? ICSE '14.

60 Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending API usage patterns. ECOOP '09.

Composing Interfering Abstract Protocols*

Filipe Militão[1], Jonathan Aldrich[2], and Luís Caires[3]

1 **Carnegie Mellon University, Pittsburgh, USA and Universidade Nova de Lisboa, Lisboa, Portugal**
`filipe.militao@cs.cmu.edu`
2 **Carnegie Mellon University, Pittsburgh, USA**
`aldrich@cs.cmu.edu`
3 **Universidade Nova de Lisboa, Lisboa, Portugal**
`lcaires@fct.unl.pt`

Abstract

The undisciplined use of shared mutable state can be a source of program errors when aliases unsafely interfere with each other. While protocol-based techniques to reason about interference abound, they do not address two practical concerns: the decidability of protocol composition and its integration with protocol abstraction. We show that our composition procedure is decidable and that it ensures safe interference even when composing abstract protocols. To evaluate the expressiveness of our protocol framework for safe shared memory interference, we show how this same protocol framework can be used to model safe, typeful message-passing concurrency idioms.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases shared memory interference, protocol composition, aliasing, linearity

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.16

1 Introduction

The interactions that can occur via shared mutable state can be a source of program errors. When different clients access the same mutable state, their actions can potentially *interfere*. For instance, the programmer may wrongly assume that a cell holds a particular type, when another part of the program has changed that cell to hold a different type. When this happens, the program may fault due to *unsafe* interference caused by unexpected actions through other aliases to that shared state. Thus, to reason about interference we must reason about how state is aliased and how the different aliases use the shared state.

Our technique builds on the use of linear capabilities [1] to track type-changing resource mutation within the framework of a linear type system. However, relying solely on linearity is often too restrictive. For instance, linearity enforces exclusive ownership of mutable state, which is incompatible with multithreading—i.e. linearity forbids sharing. To allow sharing, we extend the concept of *rely-guarantee protocols* [19]. By sequencing *steps* of

* This material is supported in part by AFRL and DARPA under agreement #FA8750-16-2-0042, and by NSA lablet contract #H98230-14-C-0140; by NOVA LINCS UID/CEC/04516/2013; and by Fundação para a Ciência e Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program under grant SFRH / BD / 33765 / 2009 and the Information and Communication Technology Institute at CMU. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of any of the sponsors.

"rely⇒guarantee" actions, each protocol characterizes an alias's local, isolated perspective on interactions with a piece of shared state:

$$\underbrace{\text{"what I } \textit{assume} \text{ about the state"} \Rightarrow \text{"what I } \textit{guarantee} \text{ about the state"}}_{\textit{current} \text{ step}} \text{ ; } \textit{next} \text{ step}$$

Since the interactions performed by an alias may change over time, a rely-guarantee protocol is formed by a sequence of steps that specify each interfering action. Each step *relies* on the shared state having some type and then, after some private actions, *guarantees* that the shared state will now have some other type, which becomes visible to other aliases. By constraining the actions of each alias, we can make strong assumptions about the kind of interference that an alias may produce, in the spirit of rely-guarantee reasoning [13]. Naturally, not all protocols compose safely. While a protocol describes its own actions on a piece of shared state, protocol *composition* will ensure that those actions are safe w.r.t. the actions that can be done via other existing (and even future) protocols over that state. Composition is safe only if the set of protocols accounts for all possible run-time action interleavings that may occur on that shared state.

Our main contribution is a decidable protocol composition procedure that also allows abstract protocols to be composed. We break down our contributions as follows:

- We adapt the existing constructs of rely-guarantee protocols [19] to work in a system with concurrent run-time semantics, and show that rely-guarantee protocols are useful to reason about safe interference in the concurrent setting.
- We give an axiomatic definition of protocol composition. We show that this procedure can be implemented in a sound and complete (w.r.t. the formal definition) algorithm that terminates on all legal inputs.[1] This algorithm is implemented in a prototype.[2]
- We show that our use of type abstraction and bounded quantification at the protocol level enables us to model new, and more general, polymorphic forms of safe modular shared state interactions.
- We prove our system sound through progress and preservation theorems that show the absence of unsafe interference in correctly typed programs. Our design ensures *memory safety* and *data-race freedom*, where linear resources are shared via protocol composition (a partial commutative monoid [16, 6]).
- We evaluate the expressiveness of our system by discussing how our core shared memory protocol framework is capable of expressing safe, typeful message-passing idioms.

Next, we briefly introduce the language that "hosts" our protocols, with the remaining text focused on discussing new protocol-level features. Sections 2 and 3 introduce our novel definition of protocol composition and its extensions to support abstract protocols. Section 4 discusses technical results; followed by discussions of expressiveness, related work, and conclusions.

1.1 Preliminaries: Language Overview

Our language supports fork/join concurrency combined with lock-based mutual exclusion, where all threads share a common heap. We use the variant of the polymorphic λ-calculus

[1] We have not proven the decidability of the entire type system, but only of the protocol composition algorithm which is at its core. The remainder of the type system is more conventional and we did not encounter difficulties with decidability when implementing similar rules in our prior work [19].

[2] See: `http://www.cs.cmu.edu/~foliveir/protocol-composition.html`

$x \in$ Variables $\quad$ $\mathtt{t} \in$ Tags $\quad$ $\mathtt{f} \in$ Fields $\quad$ $\rho \in$ Location Constants

e	::=	v	(value)			lock $\overline{v}$	(lock locations)
	|	$v.\mathtt{f}$	(field selection)		|	unlock $\overline{v}$	(unlock locations)
	|	$v\ v$	(application)		|	fork e	(spawn thread)
	|	let $x = e$ in e end	(let)				
	|	new v	(cell creation)	v	::=	ρ	(address)
	|	delete v	(cell deletion)		|	x	(variable)
	|	$!v$	(dereference)		|	$\lambda x.e$	(function)
	|	$v := v$	(assign)		|	$\{\overline{\mathtt{f} = v}\}$	(record)
	|	case v of $\overline{\mathtt{t}\#x \to e}$ end	(case)		|	$\mathtt{t}\#v$	(tagged value)

Notes: $\overline{Z}$ is a potentially empty sequence of Z elements. ρ is not source-level.

Figure 1 Grammar — Values (v) and expressions (e).

shown in Figure 1. For convenience, the grammar is let-expanded so that all constructs, except let, are defined over values. The language includes first-class functions (λ), records ($\{\overline{\mathtt{f} = v}\}$) that label a $v$alue as $\mathtt{f}$, and tagged values ($\mathtt{t}\#v$) to mark a $v$alue with a $\mathtt{t}$ag. Standard constructs are used for field selection, application, let blocks, memory allocation, deletion, assignment, dereference, and case analysis. "lock $\overline{v}$" atomically locks a non-empty set of locations (ensuring both mutual exclusion and forbidding re-entrant uses) and analogously with "unlock $\overline{v}$". "fork e" executes the $e$xpression in a new thread while sharing access to the common global heap. The operational semantics are standard and, as such, are only shown in the companion *Technical Report* [21]. They produce the standard evaluation of the language's constructs such as creating or deleting memory, spawning new threads, etc.

We type mutable references by following the design proposed in $\mathbf{L}^3$ [1]. Therefore, a mutable cell is decomposed into two components: a pure *reference*, which can be freely copied; and a linear [10] *capability*, a resource that is used to track the contents of that cell. To link a reference to its respective capability, we use location-dependent types. For instance, a new cell has type $\exists l.(\ (!\mathbf{ref}\ l) :: (\mathbf{rw}\ l\ A)\ )$. This type abstracts the fresh location, l, that was created by the memory allocation. Furthermore, we are given a reference of type "$!\mathbf{ref}\ l$" to mean a pure/duplicable (!) **ref**erence to a location l, where the information about the contents of that location is stored in the linear capability for l. The permission to access (e.g. dereference) the contents of a cell requires both the reference and the capability to be available. Our capabilities follow the format "$\mathbf{rw}\ l\ A$", meaning a **r**ead-**w**rite capability to a location l that currently has contents of type A (the type of the value, given in "new v", that initializes the new cell). We depart from [1] by making capabilities typing artifacts that only exist at the level of typing. Consequently, capabilities are managed implicitly by the type system rather than manually manipulated by the programmer via language constructs. However, we may still need to associate a capability with another type. For this reason, we use the notion of *stacking* [20]. In $\exists l.(\ (!\mathbf{ref}\ l) :: (\mathbf{rw}\ l\ A)\ )$ we see that the capability to l is stacked on top of "$!\mathbf{ref}\ l$" since the capability is to the right of the "::". This allows the capability to be bundled together with the **ref** type, but no action is required to unbundle them if they are needed separately. We refer to prior work [16, 20, 19, 1] for more details on the use of capabilities, locations, and stacking, as well as convenient abbreviations. Here, it suffices to assume that they are handled automatically by the type system, as our focus here is on safely sharing the linear resources.

In the scheme above, all the variables that reference the same location also share a single linear (i.e. "exclusively owned" or "unique") capability that tracks the changes to that

```
// assume 'y' in scope
y := "ok!";
let x = y in
  x := false;
  delete x;
  !y // Type Error: missing capability to location 'l'.
```

```
          y :!ref l, rw l int
          y :!ref l, rw l string
x :!ref l, y :!ref l, rw l string
x :!ref l, y :!ref l, rw l boolean
 x :!ref l, y :!ref l
```

Figure 2 Tracking linear capabilities.

$X \in$ Type Variables $\quad l \in$ Location Variables

$p ::= \rho \mid l \qquad u ::= l \mid X \qquad U ::= p \mid A$

$A ::=$	$!A$	(pure/persistent)	$\mid$	$(\mathbf{rec}\ X(\overline{u}).A)[\overline{U}]$	(recursive type)
$\mid$	$A \multimap A$	(linear function)	$\mid$	$A \oplus A$	(alternative)
$\mid$	$[\overline{\mathtt{f} : A}]$	(record)	$\mid$	$A \,\&\, A$	(intersection)
$\mid$	$\sum_i \mathtt{t}_i \# A_i$	(tagged sum)	$\mid$	$\mathbf{rw}\ p\ A$	(capability to p)
$\mid$	$\forall l.A$	(universal location)	$\mid$	$\mathbf{none}$	(empty resource)
$\mid$	$\exists l.A$	(existential location)	$\mid$	$\mathbf{top}$	(top)
$\mid$	$\forall X <: A.A$	(bounded universal type)	$\mid$	$A :: A$	(stacking)
$\mid$	$\exists X <: A.A$	(bounded existential type)	$\mid$	$A * A$	(separation)
$\mid$	$\mathbf{ref}\ p$	(reference type)	$\mid$	$A \Rightarrow A$	(rely)
$\mid$	$X[\overline{U}]$	(type variable)	$\mid$	$A\,;\,A$	(guarantee)

Notes: we simplify $X[]$ to X; $\oplus$, $\&$, $*$, $+$ are commutative and associative.

Figure 3 Grammar — Types (A) (Continued from Figure 1.).

location's contents (as shown in Figure 2). However, this tracking relies on a compile-time approximation of how variables alias, which constrains how state can be used. Since the linear capability must be (linearly) threaded through the program, this scheme forbids aliasing idioms that require "simultaneous" access to aliased state, such as when multiple threads share access to a cell. To enable this form of sharing, we split a linear resource into multiple protocols. Each protocol controls how an alias interacts with the shared state, without depending on precise knowledge of which variables alias each other. We will continue with a brief presentation of the base language, before diving into the details of our sharing mechanism in Section 2.

(Note that rely and guarantee types will only be discussed in the next section, when we present sharing). Our types (Figure 3) follow the connectives of linear logic [10]. For this reason a function type uses $\multimap$ (instead of $\rightarrow$) to denote a linear function. The linear restriction can be lifted when the type is preceded by a "bang", such as in $!A$, which denotes a pure/duplicable type. Records are typed as $[\overline{\mathtt{f} : A}]$ where each field $\mathtt{f}$ types the value of the record with some type A. $\sum_i \mathtt{t}_i \# A_i$ denotes a single tagged type or a sequence of tagged types separated by $+$ (such as "$\mathtt{a} \# A + \mathtt{b} \# B + \mathtt{c} \# C$"). We have separate existential and universal quantification over locations and types, since locations and types are of different kinds. Note that we leave $\forall/\exists$ as typing artifacts and as such they do not have corresponding constructs in the language. Quantification over types can provide a type bound (on the right of $<:$) and where **top** is assumed by default when the bound is omitted.

Our recursive types (assumed to be non-bottom types) are equi-recursive, interpreted co-inductively, and satisfy the usual folding/unfolding principle:

$$(\mathbf{rec}\ X(\overline{u}).A)[\overline{U}] = A\{(\mathbf{rec}\ X(\overline{u}).A)/X\}\{\overline{U}/\overline{u}\} \qquad \text{(EQ:Rec)}$$

Recursive types may include a list of type/location parameters ($\overline{u}$) that are substituted by some type/location ($\overline{U}$) on unfold, besides unfolding the recursive type variable (X).

$\boxed{\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1}$ **Typing rules, (t:*)**

(T:Pure)
$$\frac{\Gamma \mid \cdot \vdash v : A \dashv \cdot}{\Gamma \mid \cdot \vdash v : !A \dashv \cdot}$$

(T:Pure-Elim)
$$\frac{\Gamma, x : A_0 \mid \Delta_0 \vdash e : A_1 \dashv \Delta_1}{\Gamma \mid \Delta_0, x : !A_0 \vdash e : A_1 \dashv \Delta_1}$$

(T:Frame)
$$\frac{\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1}{\Gamma \mid \Delta_0, \Delta_2 \vdash e : A \dashv \Delta_1, \Delta_2}$$

(T:Function)
$$\frac{\Gamma \mid \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma \mid \Delta \vdash \lambda x.e : A_0 \multimap A_1 \dashv \cdot}$$

(T:Application)
$$\frac{\Gamma \mid \Delta_0 \vdash v_0 : A_0 \multimap A_1 \dashv \Delta_1 \qquad \Gamma \mid \Delta_1 \vdash v_1 : A_0 \dashv \Delta_2}{\Gamma \mid \Delta_0 \vdash v_0\ v_1 : A_1 \dashv \Delta_2}$$

(T:New)
$$\frac{\Gamma \mid \Delta_0 \vdash v : A \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash \mathsf{new}\ v : \exists l.((!\mathbf{ref}\ l) :: (\mathbf{rw}\ l\ A)) \dashv \Delta_1}$$

(T:Delete)
$$\frac{\Gamma \mid \Delta_0 \vdash v : \exists l.((!\mathbf{ref}\ l) :: (\mathbf{rw}\ l\ A)) \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash \mathsf{delete}\ v : \exists l.A \dashv \Delta_1}$$

(T:Assign)
$$\frac{\Gamma \mid \Delta_0 \vdash v_1 : A_0 \dashv \Delta_1 \qquad \Gamma \mid \Delta_1 \vdash v_0 : \mathbf{ref}\ p \dashv \Delta_2, \mathbf{rw}\ p\ A_1}{\Gamma \mid \Delta_0 \vdash v_0 := v_1 : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ A_0}$$

(T:Let)
$$\frac{\Gamma \mid \Delta_0 \vdash e_0 : A_0 \dashv \Delta_1 \qquad \Gamma \mid \Delta_1, x : A_0 \vdash e_1 : A_1 \dashv \Delta_2}{\Gamma \mid \Delta_0 \vdash \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\ \mathsf{end} : A_1 \dashv \Delta_2}$$

(T:Dereference-Linear)
$$\frac{\Gamma \mid \Delta_0 \vdash v : \mathbf{ref}\ p \dashv \Delta_1, \mathbf{rw}\ p\ A}{\Gamma \mid \Delta_0 \vdash !v : A \dashv \Delta_1, \mathbf{rw}\ p\ ![]}$$

(T:LocOpenBind)
$$\frac{\Gamma, l : \mathbf{loc} \mid \Delta_0, x : A_1 \vdash e : A_2 \dashv \Delta_1}{\Gamma \mid \Delta_0, x : \exists l.A_1 \vdash e : A_2 \dashv \Delta_1}$$

(T:Subsumption)
$$\frac{\Gamma \vdash \Delta_0 <: \Delta_1 \qquad \Gamma \mid \Delta_1 \vdash e : A_0 \dashv \Delta_2 \qquad \Gamma \vdash A_0 <: A_1 \qquad \Gamma \vdash \Delta_2 <: \Delta_3}{\Gamma \mid \Delta_0 \vdash e : A_1 \dashv \Delta_3}$$

Note: bounded variables of a construct and of quantifiers must be fresh in the rule's conclusion.

Figure 4 Typing rules (selected, see [21] for complete set).

We use $\oplus$ to denote a union of alternative types, and & to denote a linear choice of different types. **none** is the empty resource. Finally we have "$A_0 :: A_1$" for stacking resource A_1 on top of A_0. Stacking is not commutative, so that it is not guaranteed that "$A_0 :: A_1 :: A_2$" can be used in place of "$A_0 :: A_2 :: A_1$". To enable resource commutation, we use the $*$ operator such that "$A_0 :: (A_1 * A_2)$" and "$A_0 :: (A_2 * A_1)$" are interchangeable via subtyping. For clarity, we will review these type annotations as we present examples further below. Note that we do not syntactically distinguish resources (such as capabilities or protocols) from value-inhabited types. However, the type system ensures that types such as **none** can never be used to type a value. Indeed, even though "wrong" types can be assumed (such as in a function's argument) they can never actually be introduced as values.

To enable automatic threading of resources, we use a type-and-effect system with judgments of the form: $\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1$ stating that with lexical environment Γ and linear resources Δ_0 we assign the expression e the type A, with effects resulting in the resources in Δ_1. The typing environments are defined as follows:

Γ	$::=$	$\cdot$		(empty)	Δ	$::=$	$\cdot$	(empty)
	$\mid$	$\Gamma,\ x : A$		(variable binding)		$\mid$	$\Delta,\ x : A$	(linear binding)
	$\mid$	$\Gamma,\ p : \mathbf{loc}$		(location assertion)		$\mid$	$\Delta,\ A$	(linear resource)
	$\mid$	$\Gamma,\ X <: A$		(bound assertion)				
	$\mid$	$\Gamma,\ X : k$		(kind assertion)	k	$::=$	$\mathbf{type} \mid \mathbf{type} \to k \mid \mathbf{loc} \to k$	(kinds)

Recursive type variables are given an $\to$ kind, where the left hand side tracks the type/location kind of a parameter of that recursive type.

Figure 4 includes a few selected typing rules. Additional rules are shown below as they become relevant to the discussion on sharing, with the remainder left to the T.R.. (T:Pure) types a value as pure if the value does not use any resources. If a variable is of a pure type, then (T:PureElim) allows the binding to be moved to the linear context with its type explicitly "banged" with !. (T:Frame) enables framing [27] resources that are not used by an expression, just threaded through the expression. Since a function, (T:Function), can depend on the resources inside of Δ (which the function captures), a functional value must be linear. However, the function can later be rendered pure (!) through the use of (T:Pure) if the set of resources it captures is actually empty. (T:Application) is the standard rule. As discussed above, (T:New) and (T:Delete) manipulate types that abstract the underlying location that was created or that is to be deleted. (T:Assign) updates the contents of a location with the type of the newly assigned value. (T:Let) threads the effects of e_0 to the initial linear resources of e_1, sequencing the evaluation of the expressions as usual. (T:Dereference-Linear) removes the contents of a cell, leaving the residual "unit" type behind (the semantics leave the cell unchanged but unusable through typing). (T:LocOpenBind) enables non-syntax-directed opening of existential location packages.

The subtyping rules are deferred to the T.R., but it suffices to know the subtyping judgment, $\Gamma \vdash A_0 <: A_1$, which states that A_0 is a subtype of A_1, meaning that A_0 can be used anywhere A_1 is expected. An analogous judgment governs subtyping between linear environments, $\Gamma \vdash \Delta_0 <: \Delta_1$. Thus, the (T:Subsumption) rule simply states that we can type an expression while using weaker assumptions and ensuring a stronger result and effect, as these types cannot break the conclusion's type expectations.

2 A Protocol for Modeling Fork-Join Interactions

We begin by describing how non-abstracted protocols compose and how rely-guarantee protocols work in the concurrent setting. Our language supports the fork/join model of concurrency, in which a join is encoded via shared state interactions. There are two participants in this interaction: the Main thread and the Forked thread. The forked thread computes some *result*. When the main thread joins the forked thread it will *wait* until the result becomes available, if it is not yet ready. Our primitives to interact with shared state are reading/writing and locking/unlocking. Because of this, our protocols must explicitly model the "wait for result" cycle of a join.[3] A thread scheduler could reduce or eliminate the spinning caused by this "busy-wait", but this is beyond the scope of our discussion. We define the two protocols as:

$$\texttt{F} \triangleq \mathsf{Wait} \Rightarrow \mathsf{Result}\ ;\ \textbf{none} \qquad \texttt{M} \triangleq (\mathsf{Wait} \Rightarrow \mathsf{Wait}\ ;\ \texttt{M}) \oplus (\mathsf{Result} \Rightarrow \mathsf{Done}\ ;\ \mathsf{Done})$$

Each protocol contains a sequence of steps that control the use of locks and specify the (type) assumptions on that locked state. Since locks hide all private actions, the protocols will only need to model the changes that become visible upon unlocking. These changes are bounded by a single lock-unlock block, which is mapped to a single rely⇒guarantee step in the protocol. When we lock a cell we will *assume* that the state is of some type and, when we eventually unlock that cell, we will *guarantee* that it changed to some other type. Multiple steps can be sequenced using the ; operator.

The forked thread will be given the F protocol. This protocol initially assumes that the shared state is of type Wait on locking. In order to legally unlock that cell, we must

[3] Each protocol must be aware of all valid states, as an omission would leave room for unsafe interference, such as when later re-splitting that protocol.

first fulfill the obligation to mutate the state to Result. Once that guarantee is obeyed the protocol continues as **none**. This empty resource type models termination since the forked thread will never be able to access that shared state again. Note that since subsequent steps may be influenced by the guarantee of the current step, a protocol step is to be interpreted as "Wait $\Rightarrow$ (Result ; **none**)".

The `Main` protocol includes two alternative ($\oplus$) steps that describe different uses of the shared state. If we find the shared cell containing the Wait type then the main thread must leave the state with the same type, before later retrying `M`. Otherwise, if we find the cell containing a Result, we know that `F` has already terminated and can no longer access the shared state. In that situation, we mutate the cell to Done and unlock it so that each lock always has a matching unlock. Afterwards, the protocol continues as Done, a type that is just a regular linear capability. Thus, `M` recovered ownership of the shared state and Done can continue to be used without locking since the cell is no longer shared. We can now give concrete definitions for Wait, Result, and Done as types describing a single capability to location l as follows:

$$\mathsf{Wait} \triangleq \mathbf{rw}\ l\ \texttt{Wait\#}![] \qquad \mathsf{Result} \triangleq \mathbf{rw}\ l\ \texttt{Result\#int} \qquad \mathsf{Done} \triangleq \mathbf{rw}\ l\ ![]$$

Wait is a capability to location l containing a tagged value, where `Wait` is the tag and "![]" (a pure empty record) is the type of the value. Result is a capability for l containing an integer value tagged with `Result`. The two tags will enable us to distinguish between the Wait and Result alternatives by using standard case analysis. With Done the content is an empty pure record ("unit").

Each protocol describes an alias's local, isolated view of the evolution of the shared state. Thus, we can discuss the uses of each protocol independently. Because a protocol is a linear resource, the forked thread will "consume" or "capture" `F` in its context, making it unavailable to the main thread. As with any linear resource, `F` is tracked by the linear typing environment (Δ) and is either used by an expression or threaded through to the next expression. However, the forked thread and main thread can share the enclosing lexical typing environment (Γ) because it only contains pure/duplicable assumptions. A possible use of the `F` protocol follows.

```
3  fork
4    let r = work {} in
5      lock c;
6      c := Result#r;
7      unlock c
8    end
```

$\Gamma = \texttt{c} : \mathbf{ref}\ l, l : \texttt{loc}$	$\mid \Delta = \texttt{work} : ![] \multimap \texttt{int}, \texttt{F}$
$\Gamma = \texttt{r} : \texttt{int}, \ldots$	$\mid \Delta = \mathsf{Wait} \Rightarrow (\mathsf{Result}\ ;\ \mathbf{none})$
$\Gamma = \ldots$	$\mid \Delta = \mathsf{Wait}, (\mathsf{Result}\ ;\ \mathbf{none})$
$\Gamma = \ldots$	$\mid \Delta = \mathsf{Result}, (\mathsf{Result}\ ;\ \mathbf{none})$
$\Gamma = \ldots$	$\mid \Delta = \mathbf{none}$
$\Gamma = \ldots$	$\mid \Delta = \cdot$

Γ contains a reference (`c`) to the location (l) that is being shared by the protocol, and Δ contains a variable with the (linear) function that computes the `work` that the thread will do. (In this example both protocols refer to a well-known common location, but our technique also allows each protocol to $\exists$ abstract its locations.) Line 4 consumes the function `work` by calling it and storing the result in variable `r`. At this point we want to update the shared state to signal that the result is ready. Since we are accessing shared state in a multi-threaded environment we first lock the shared location that is being referenced by `c`. To type a lock we must map the locations listed in the lock to those contained in the rely type of the protocol. Well-formedness conditions on the protocols ensure that, at each step, the rely and the guarantee types refer the same set of locations so that no lock on a location

goes without a respective unlock (later on).

$$\frac{\overline{\Gamma \mid \cdot \vdash v : \mathbf{ref}\ p \dashv \cdot} \qquad \mathbf{locs}(A_0) = \overline{p}}{\Gamma \mid \Delta,\ A_0 \Rightarrow A_1 \vdash \mathsf{lock}\ \overline{v} : ![] \dashv \Delta, A_0, A_1} \text{ (T:Lock-Rely)}$$

When locking (line 5), the step of `F` is broken down into its two components: the rely type (Wait) and the guarantee type (Result; **none**). While Wait describes the linear resources that are now available to use, the guarantee type is an obligation to mutate the state to fulfill the given type before unlocking. Indeed, line 7 is only valid because the shared state was modified to match the promised guarantee type (Result).

$$\frac{\overline{\Gamma \mid \cdot \vdash v : \mathbf{ref}\ p \dashv \cdot} \qquad \mathbf{locs}(A_0) = \overline{p}}{\Gamma \mid \Delta,\ A_0, (A_0; A_1) \vdash \mathsf{unlock}\ \overline{v} : ![] \dashv \Delta, A_1} \text{ (T:Unlock-Guarantee)}$$

(with parenthesis used for clarity). Once the guarantee is fulfilled, we can move on to the next step of the protocol (in the case of `F`, **none**; or A_1, in the case of the rule above). The **none** type is the empty resource that can be automatically discarded, leaving Δ empty ($\cdot$). Thus, the uses of protocols are mapped to the (T:Lock-Rely) and (T:Unlock-Guarantee) rules that step a protocol. We now show the rest of the encoding:

```
1 let newFork = λwork.
2   let c = new Wait#{} in
3     fork ... // lines 3 to 8 shown above.
```

$\Gamma = \cdot$	$\mid \Delta =$ `work` $: ![] \multimap$ `int`
$\Gamma =$ `c` $:$ `ref` $l, l :$ `loc`	$\mid \Delta =$ `rw` l `Wait#![]`, ...
$\Gamma =$ ...	$\mid \Delta =$ `M`, `F`, ...

To simplify the presentation, our term language is stripped of type annotations. However, the newFork function has type $!(\ (\ ![] \multimap$ `int` $)\ \multimap\ (\ ![] \multimap$ `int` $)\ )$ where the argument of this pure function is the `work` to be done by the thread, as was shown above. The resulting function is the join (shown below) that, once called, waits for the forked thread's result. Line 2 creates the cell that will be shared by the main and forked threads. This new cell, although typed $\exists l.(\ (!\mathbf{ref}\ l) :: (\mathbf{rw}\ l$ `Wait#![]`$)\ )$, is automatically opened by the type system via (T:LocOpenBind) to allow direct access to the **ref** l reference via variable `c`.

Line 3 shares the cell by splitting the capability to location l into the `M` and `F` protocols. This split is done in a non-syntax-directed way through (T:Subsumption) (of Figure 4), combined with the following rule for subtyping on Δ's:

$$\frac{\Gamma \vdash \Delta_0 <: \Delta_1 \qquad \Gamma \vdash A_0 \Rrightarrow A_1 \mid\mid A_2}{\Gamma \vdash \Delta_0, A_0 <: \Delta_1, A_1, A_2} \text{ (SD:Share)}$$

Where the following resource split ($\Rrightarrow$) is used:

$$\Gamma \ \vdash\ \mathsf{Wait} \ \Rrightarrow\ \mathtt{M} \mid\mid \mathtt{F} \qquad\qquad (\text{recall: } \mathsf{Wait} \triangleq \mathbf{rw}\ l\ \mathtt{Wait\#![]})$$

This split results in the capability to location l being replaced by the two protocols, `M` and `F`, in Δ. The composition check (described in the next subsection) relies on the knowledge that `M` and `F` share the same location. Once the protocols are known to compose safely, however, we no longer need to track this sharing—each protocol can abstract the location being accessed under a different name, and they can be used independently. The fork expression is typed by consuming the resources that the fork will use (such as `F` in the fork of line 3):

$$\frac{\Gamma \mid \Delta \vdash e : ![] \dashv \cdot}{\Gamma \mid \Delta \vdash \mathsf{fork}\ e : ![] \dashv \cdot} \text{ (T:Fork)}$$

This rule is somewhat similar to (T:Function), but the result type is unit because fork does not produce a result. Thus, a fork is executed for the effects it produces on the shared state.

As such, to avoid leaking resources, the final residual resources of the forked expression must be empty and the resulting value pure (note that "$!A <: ![]$").

Finally, we show the join function that will "busy-wait" for the forked thread to produce a result. Its use of both **rec**ursion and case analysis should be straightforward as they follow standard usage. The following text will focus on the less obvious details.

```
9   λ_.rec R.                                  Δ = ( Wait ⇒ (Wait; M) )  ⊕  ( Result ⇒ (Done; Done) )
10                                             [a]Δ = Wait ⇒ (Wait; M)     [b]Δ = Result ⇒ (Done; Done)
11      lock c;                                [a]Δ = Wait, (Wait; M)      [b]Δ = Result, (Done; Done)
12      case !c of                             [a]Δ = rw l ![], (Wait; M)  [b]Δ = rw l ![], (Done; Done)
13        Wait#x → // must restore linear value                [a]Δ = rw l ![], (Wait; M)
14          c := Wait#x;                                       [a]Δ = Wait, (Wait; M)
15          unlock c;                                          [a]Δ = M
16          R // retries
17      | Result#x →                           [b]Γ = x : int, ...   | Δ = rw l ![], (Done; Done)
18          unlock c;                          [b]Γ = x : int, ...   | Δ = rw l ![]
19          delete c;                          [b]Γ = x : int, ...   | Δ = ·
20          x
21      end
22    end
```

We omit Γ to center the discussion on the contents of Δ. The alternative type (⊕) lists a union of types that may be valid at that point in the program. To use such a type, an expression must consider each alternative individually via (T:Alternative-Left):

$$\frac{\Gamma \mid \Delta_0, A_0 \vdash e : A_2 \dashv \Delta_1 \qquad \Gamma \mid \Delta_0, A_1 \vdash e : A_2 \dashv \Delta_1}{\Gamma \mid \Delta_0, A_0 \oplus A_1 \vdash e : A_2 \dashv \Delta_1} \text{ (T:Alternative-Left)}$$

The breakdown of ⊕ (line 10) is done automatically by the type system. Thus, the body of the recursion must be typed individually under each one of those alternatives, marked as `[a]` and `[b]`. The type of the resource on each alternative contains a sum type that matches different branches in the case of line 12. Note that it is safe for this sum type to only match a subset of the branches that the case lists. The remaining branches are simply ignored when typing the case with that sum type:

$$\frac{\Gamma \mid \Delta_0 \vdash v : \sum_i \mathtt{t}_i \# A_i \dashv \Delta_1 \qquad \overline{\Gamma \mid \Delta_1, x_i : A_i \vdash e_i : A \dashv \Delta_2} \qquad i \leq j}{\Gamma \mid \Delta_0 \vdash \mathsf{case}\ v\ \mathsf{of}\ \overline{\mathtt{t}_j \# x_j \to e_j}\ \mathsf{end} : A \dashv \Delta_2} \text{ (T:Case)}$$

This enables the same case to produce different effects, such as obeying incompatible guarantees, based solely on the tagged contents of v. For instance, the `Result` branch will recover ownership and destroy the shared cell (line 19), while the `Wait` branch must restore the linear value of that cell (that was removed by the linear dereference of line 12, that left "**rw** l $![]$" in Δ) before retrying. Although line 19 deletes the cell, we first unlock the cell to fulfill the "Done; Done" guarantee of the final protocol step.

A rely-guarantee protocol is a specification of each lock-unlock usage, modeled by a protocol type. Therefore, we will continue the discussion on interference by only looking at the protocols, while omitting the actual concrete programs that use them.

2.1 Checking Safe Protocol Composition

We now introduce our main contribution: a novel axiomatic definition of protocol composition, which is later extended to support abstraction. Composing protocols over some shared state requires considering all possible ways in which the use of these protocols may be interleaved. Thus, regardless of the non-deterministic way by which aliases are interleaved at run-time, a correct composition will ensure that all possible uses are safe.

Intuitively, a binary protocol split will generate an infinite binary tree representing all combinations of interleaved uses of the two new protocols. Each node of that tree has two children based on which protocol remains stationary while the other is stepped. Since this tree may be infinite, we must build a co-inductive proof of safe interference. We only consider binary splits when checking composition but since a protocol can be later re-split, there is no limit to how many protocols may share some state.

The two protocols, `M` and `F`, shown above contain a finite number of different positions. We call a *configuration* the combination of the positions of each protocol and the current type of the shared resources. Each configuration is of the form:

$$\langle\ \Gamma \vdash \mathit{Resources} \Rrightarrow \mathit{Protocol} \mid\mid \mathit{Protocol}\ \rangle$$

Thus, when we split a Wait cell into protocols `M`[4] and `F`[5], we get the following set of configurations that simulate the uses done via the protocols (seen as atomic public transitions of lock-unlock uses, corresponding to the respective rely and guarantee types):

$$\{\ \ ❶\ \langle\ \Gamma \vdash \mathsf{Wait} \Rrightarrow \mathtt{M} \mid\mid \mathtt{F}\ \rangle\ ,\ ❷\ \langle\ \Gamma \vdash \mathsf{Result} \Rrightarrow \mathtt{M} \mid\mid \textbf{none}\ \rangle\ ,$$
$$\quad ❸\ \langle\ \Gamma \vdash \mathsf{Done} \Rrightarrow \mathsf{Done} \mid\mid \textbf{none}\ \rangle\ ,\ ❹\ \langle\ \Gamma \vdash \textbf{none} \Rrightarrow \textbf{none} \mid\mid \textbf{none}\ \rangle\ \}$$

Configuration ❶ represents the initial split of Wait into `M` and `F`. Starting from some configuration, we will leave one of the protocols stationary while we simulate a use of the shared state (a *step*) with the remaining protocol. From ❶ if we step `M` we will stay in the same configuration. If instead `F` is stepped, we get to configuration ❷ that changed the state to Result and terminates the `F` protocol. By continuing to step `M` we have the two last configurations: ❸ where the last step of `M` is ready to recover ownership, and ❹ where the ownership of the shared resource was recovered and all protocols have terminated (i.e. all resources are empty, **none**).

Upon sharing, the ownership of the shared resources belongs to all intervening protocols; all protocols can access the shared resources through locking. Ownership recovery means that this ownership is given back to one single protocol and "revoked" from all remaining protocols. In our protocols, recovery is modeled via protocol termination, such that a step transitions to a state rather than to another protocol step. However, to be safe, we must be sure that this permanent ownership transfer only occurs on the *last* protocol to terminate, ensuring that no other protocol may accidentally assume that that shared state is still available. The ownership recovery in ❸ transfers Done from the "pool" of shared resources to the alias that uses the last step of the `M` protocol. We also see by ❹ that this stepping consumes both the shared resource (leaving it as **none**) and the final "step" of `M` (leaving the protocol position also as **none**).

All protocol configurations shown above can take a step. (Even **none** can take a vacuous step that remains in the same configuration since **none** cannot change the shared resources.) Therefore, each protocol will always find an expected state in the shared cell regardless of how protocols are interleaved—i.e. all interference is safe since no configuration is stuck. A stuck configuration occurs when at least one of the protocols cannot take a step with the current type of the shared resources. For instance, $\langle \Gamma \vdash \mathsf{Result} \Rrightarrow \mathtt{M} \mid\mid \mathtt{F} \rangle$ cannot take a step with `F` since `F` does not rely on Result in any of its available steps. If such stuck configurations were allowed to occur, then a program could fault due to unexpected values stored in shared cells

[4] $\mathtt{M} \triangleq (\ \mathsf{Wait} \Rightarrow (\ \mathsf{Wait}\ ;\ \mathtt{M}\ )\ ) \oplus (\ \mathsf{Result} \Rightarrow (\ \mathsf{Done}\ ;\ \mathsf{Done}\ )\ )$
[5] $\mathtt{F} \triangleq \mathsf{Wait} \Rightarrow (\ \mathsf{Result}\ ;\ \textbf{none}\ )$

$$
\begin{array}{rcl}
P, Q & ::= & (\textbf{rec}\ X(\overline{u}).P)[\overline{U_P}] \mid X[\overline{U_P}] \mid P \oplus P \mid P \,\&\, P \mid \textbf{none} \\
 & \mid & S \Rightarrow P \mid S\,;\, P \mid \exists l.P \mid \forall l.P \mid \exists X <: A.P \mid \forall X <: A.P \\
S & ::= & (\textbf{rec}\ X(\overline{u}).S)[\overline{U_S}] \mid X[\overline{U_S}] \mid S \oplus S \mid S \,\&\, S \mid \textbf{none} \mid A * A \mid \textbf{rw}\ p\ A \\
R & ::= & P \mid S
\end{array}
$$

Note: that the structure of allowed protocols is further restricted via protocol composition, beyond the syntactical categories above. Namely, abstraction is only enabled by the rules of Section 3.3.

Figure 5 Grammar for checking safe protocol composition: *P*rotocols, *S*tates, and *R*esources.

or due to attempts to access cells that were destroyed using wrong assumptions of ownership recovery.

Protocol composition ensures that a resource, R (capabilities or protocols), can be shared (split) as two protocols, P and Q, noted: $\Gamma \vdash R \Rrightarrow P \mid\mid Q$. Figure 5 lists the grammatical categories (for protocols, states and resources) that we consider when composing protocols. As exemplified above we use a set of *configurations*, C, to represent the positions of each protocol as we traverse all possible interleaved uses of the two new protocols. C is defined as:

$$C \quad ::= \quad C \cdot C \quad \text{(union)} \qquad \mid \quad \langle\, \Gamma \vdash R \Rrightarrow P \mid\mid Q \,\rangle \quad \text{(configuration)}$$

Protocol composition, applied via (SD:SHARE), ensures that all configurations reachable through stepping are themselves able to take a step, as follows:

$$\frac{\langle\, \Gamma \vdash R \Rrightarrow P \mid\mid Q \,\rangle\uparrow}{\Gamma \vdash R \Rrightarrow P \mid\mid Q}\ \text{(WF:SPLIT)} \qquad\qquad \frac{C_0 \mapsto C_1 \qquad C_1\uparrow}{C_0\uparrow}\ \text{(WF:CONFIGURATION)}$$

Where $C\uparrow$ signals the *divergence* of stepping, consistent with the co-inductive nature of protocol composition. We use a double line, as in (WF:CONFIGURATION), to mean that a rule is to be interpreted co-inductively. This definition accounts for protocols that never terminate and also ensures that all protocols can take a step with a given resource.

We now discuss the basic protocol composition definition of Figure 6. (C:ALLSTEP) synchronously steps all existing configurations, where each configuration is stepped through (C:STEP). We use $\mathcal{R}_*$ (where $_*$ is either L or R) to specify the configuration reduction context on one of the protocols of a configuration, while the remaining one remains stationary, i.e.:

$$
\begin{array}{rcll}
\mathcal{R}_L[\Box] & = & \Box \mid\mid Q & \text{(for the } L\text{eft protocol, } Q \text{ is stationary)} \\
\mathcal{R}_R[\Box] & = & P \mid\mid \Box & \text{(for the } R\text{ight protocol, } P \text{ is stationary)}
\end{array}
$$

The subsequent stepping rules use $\mathcal{R}$ to range over both $\mathcal{R}_L$ and $\mathcal{R}_R$.

We use three distinct label prefixes to group the stepping rules based on whether a rule is stepping over a protocol (C-PS:*), stepping over some state (C-SS:*), or is applicable on both kinds of resource (C-RS:*). (C-RS:NONE) "spins" a configuration since a terminated protocol cannot use the shared resources but must be stuck-free for consistency with our definition. The following (C-RS:*ALTERNATIVE) and (C-RS:*INTERSECTION) rules "dissect" a resource based on the alternative ($\oplus$) or choice ($\&$) presented. Each different alternative state must be individually considered by a protocol, while only one alternative step of a protocol needs to be valid. The situation is the reverse for choices: all choices of a protocol must have a valid step, but a step of a protocol can choose which resource to consider when stepping. State stepping, (C-SS:STEP), transitions the step of the protocol and changes the state of the shared resources to reflect the guaranteed state of the protocol. Ownership recovery, (C-SS:RECOVERY), "consumes" the shared state (leaving it as **none**) which models the transfer of ownership of that state back to the client context that uses the final step

$\boxed{C \mapsto C}$ **Composition, (c:*)**

$$\frac{\langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}_L[P] \,\rangle \mapsto C_0 \qquad \mathcal{R}_L[\square] = \square \mid\mid Q \qquad \langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}_R[Q] \,\rangle \mapsto C_1 \qquad \mathcal{R}_R[\square] = P \mid\mid \square}{\langle\, \Gamma \vdash R \Rrightarrow P \mid\mid Q \,\rangle \mapsto C_0 \cdot C_1}\ (\textsc{c:Step}) \qquad \frac{C_0 \mapsto C_2 \qquad C_1 \mapsto C_3}{C_0 \cdot C_1 \mapsto C_2 \cdot C_3}\ (\textsc{c:AllStep})$$

Composition — Reduction Step, (c-rs:*)

$$\frac{}{\langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[\mathsf{none}] \,\rangle \mapsto \langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[\mathsf{none}] \,\rangle}\ (\textsc{c-rs:None})$$

$$\frac{\langle\, \Gamma \vdash R_0 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C}{\langle\, \Gamma \vdash R_0 \& R_1 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C}\ (\textsc{c-rs:StateIntersection}) \qquad \frac{\langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[P_0] \,\rangle \mapsto C}{\langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[P_0 \oplus P_1] \,\rangle \mapsto C}\ (\textsc{c-rs:ProtocolAlternative})$$

$$\frac{\langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[P_0] \,\rangle \mapsto C_0 \qquad \langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[P_1] \,\rangle \mapsto C_1}{\langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[P_0 \& P_1] \,\rangle \mapsto C_0 \cdot C_1}\ (\textsc{c-rs:ProtocolIntersection}) \qquad \frac{\langle\, \Gamma \vdash R_0 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C_0 \qquad \langle\, \Gamma \vdash R_1 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C_1}{\langle\, \Gamma \vdash R_0 \oplus R_1 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C_0 \cdot C_1}\ (\textsc{c-rs:StateAlternative})$$

Composition — State Stepping, (c-ss:*)

$$\frac{}{\langle\, \Gamma \vdash S_0 \Rrightarrow \mathcal{R}[S_0 \Rightarrow S_1; P] \,\rangle \mapsto \langle\, \Gamma \vdash S_1 \Rrightarrow \mathcal{R}[P] \,\rangle}\ (\textsc{c-ss:Step})$$

$$\frac{}{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[S] \,\rangle \mapsto \langle\, \Gamma \vdash \mathsf{none} \Rrightarrow \mathcal{R}[\mathsf{none}] \,\rangle}\ (\textsc{c-ss:Recovery})$$

Composition — Protocol Stepping, (c-ps:*)

$$\frac{}{\langle\, \Gamma \vdash S_0 \Rightarrow S_1; Q \Rrightarrow \mathcal{R}[S_0 \Rightarrow S_1; P] \,\rangle \mapsto \langle\, \Gamma \vdash Q \Rrightarrow \mathcal{R}[P] \,\rangle}\ (\textsc{c-ps:Step})$$

Figure 6 Basic protocol composition stepping rules.

of the protocol. Protocol stepping, (C-PS:STEP), requires an exact simulation of the rely and guarantee types when stepping both the simulated protocol and the current stepping protocol. Note that the rules above also enable the re-splitting of a protocol by extending an ownership recovery step. In this situation, we have that the simulation of the original protocol will seamlessly switch from the protocol stepping rules to the state stepping rules.

3 Polymorphic Protocol Composition

Up to this point, protocol composition does a strict stepping of protocols. Consequently, stepping requires each protocol to know the exact type representation of the shared resources. Ideally, to improve both locality and modularity, each protocol should only depend on the type information that is relevant to the actions done through that alias. For instance, the action done through the `F` protocol of page 6 does not need to know the precise type (Wait) that is initially stored in location l. Thus, we want to be able to abstract Wait as X such that the protocol only keeps the typing information that is relevant to that protocol's *local perspective* on the shared resources: $\exists X.(\ \texttt{rw}\ l\ X\ \Rightarrow\ (\ \texttt{rw}\ l\ \texttt{Result\#int}\ ;\ \mathsf{none}\ )\ )$. Similarly, the wait step of the `M` protocol only depends on the tag of the shared cell enabling everything else to be abstracted from its perspective: $\exists X.(\ \texttt{rw}\ l\ \texttt{Wait\#}X\ \Rightarrow\ (\ \texttt{rw}\ l\ \texttt{Wait\#}X\ ;\ \texttt{M}\ )\ )\ \oplus\ (\ldots)$.

Since rely-guarantee protocols are first-class types, they can move outside the scope of a "module". Without this form of abstraction, such a move would either expose potentially private information or limit how clients may later re-split the shared resources. While enabling protocols to abstract part of their uses based on their perspective of the shared resources improves modularity and increases flexibility, it also brings new challenges on defining safe protocol composition and ensuring its termination. We will focus the discussion on two new aliasing idioms that this kind of abstraction enables: a) *existential-universal interaction*, how a universally quantified guarantee can safely interact with an existentially quantified rely; and b) *step extensions over abstractions*, how abstractions enable existing protocol steps to be re-split (i.e. nested protocol re-splitting) yet without the risk of introducing unsafe interference on older protocols of that state. Section 4 approaches the decidability problem. The remaining of this section starts by introducing the basic intuition of how protocol-level abstraction works, before extending our definition of composition to account for abstraction.

3.1 Existential-Universal Interaction

Enabling existential abstraction over the contents of the shared state will naturally allow a greater decoupling from the actions done by other aliases to that shared state. However, since a protocol encodes *sequences* of steps, ensuring safety must also account for the validity of the scope of the opaque type. For instance, consider the composition:

$$\Gamma \vdash \mathbf{rw}\ p\ \texttt{int} \Rrightarrow \exists X.(\ \mathbf{rw}\ p\ X\ \Rightarrow\ \mathbf{rw}\ p\ X\ ;\ \mathbf{rw}\ p\ X\ \Rightarrow\ \ldots\ )\ \|\ (\ \mathbf{rw}\ p\ \texttt{int}\ \Rightarrow\ \mathbf{rw}\ p\ \texttt{boolean}\ ;\ \ldots\ )$$

On the left protocol, the scope of X extends beyond a single ($\Rightarrow$) step. Because the right protocol can change the underlying representation of X, this composition cannot be ruled safe. Indeed, if X were of a pure type, the left protocol could potentially swap X's to an X of a different representation, in a way that would unsafely interfere with the right protocol's assumptions on the precise contents of the shared state. Thus, while the left protocol depends on an opaque type, that protocol still requires that the scope/"lifetime" of X extends to the next step although the protocol does not impose any other type restrictions on X.

We now discuss the core ideas that enable the safe composition of protocols that interact over abstractions. First the interaction will only occur via the "lifetime" of the stored type (as it changes on each step), and then we will use bounded quantification to enable types that are less opaque. Consider the following protocols that are sharing a location p:

$$\begin{array}{lcl} \texttt{Nothing} & \triangleq & \exists X.(\ \mathbf{rw}\ p\ X \Rightarrow \mathbf{rw}\ p\ X\ ;\ \texttt{Nothing}\ ) \\ \texttt{Full}[Y] & \triangleq & \mathbf{rw}\ p\ Y \Rightarrow \forall Z.(\ \mathbf{rw}\ p\ Z\ ;\ \texttt{Full}[Z]\ ) \end{array}$$

The `Nothing` protocol is defined using X to abstract the contents of the shared cell on a single step, while also guaranteeing that X is restored before repeating the protocol. Thus, `Nothing` cannot publicly modify the shared state, although p can undergo private changes. Conversely, `Full` is able to arbitrarily modify the shared state by allowing its clients to pick any type to apply to the $\forall$ of the guarantee. `Full` itself is parametric on the type that is currently stored in the shared cell, Y. Each step of `Full` can exploit the precise local information on how the state was modified, by remembering its own changes to cell p. However, the "lifetime" of X in `Nothing` is restricted to a single step. Naturally, to be able to check this composition in a finite number of steps, we must check the changes done by `Full` abstractly. To illustrate how composition works in this case, consider the following split where p initially holds a value of type `int`:

$$p : \mathbf{loc} \vdash \mathbf{rw}\ p\ \texttt{int}\ \Rrightarrow\ \texttt{Full}[\texttt{int}]\ \|\ \texttt{Nothing}$$

Protocol composition results in the following set of configurations:

$$\begin{array}{l}\{\ ❶\ \langle\, p : \mathbf{loc} \qquad\quad \vdash \mathbf{rw}\ p\ \mathtt{int} \Rrightarrow \mathtt{Full[int]}\ \ ||\, \mathtt{Nothing}\ \rangle\ , \\ \ \ ❷\ \langle\, p : \mathbf{loc},\, Z : \mathbf{type} \vdash \mathbf{rw}\ p\ Z \ \ \Rrightarrow\ \mathtt{Full}[Z]\ \ ||\, \mathtt{Nothing}\,\rangle\ \}\end{array}$$

The use of abstraction will mean that each configuration may have different assumptions of type (and location) variables. Configuration ❶ is the initial configuration given by the split above, which includes the assumption that p is a known **loc**ation. To step `Nothing` from ❶, we must first find a representation type to open the existential. This type is found by unifying the current state of the shared state (**rw** p `int`) with the rely type of `Nothing` (**rw** p X). Thus, we see that X is abstracting `int`. After we open the existential, by exposing the `int` type, we see that the step will preserve `int` resulting in `Nothing` yielding the same ❶ configuration. To step ❶ with `Full`[`int`], we must consider that its resulting guarantee is abstract. The new configuration, ❷, must consider a fresh type variable to represent that new type that a client can pick. In this case, we used Z to represent that new type. It is straightforward to see that if we were to step `Nothing` from ❷ we would remain in configuration ❷ following similar reasoning to that done for ❶. Perhaps the surprising aspect is that further steps with `Full` will also yield configurations that are *equivalent* to ❷.

The typing environment plays a crucial role in enabling us to close the proof of safe composition. Although each step of `Full` must consider a fresh type due to the $\forall$, stepping results in configurations that are equivalent up to renaming of variables and weakening of Γ. Weakening allows us to ignore variables that no longer occur free in a configuration. This means that further steps with `Full` result in configurations that are equivalent to already seen configurations. Thus, although the set of different types that can be applied to `Full`'s guarantee is infinite, the number of *distinct interactions* that can legally occur through that shared state is finite if we model those interactions abstractly. Lifetime conflicts cannot occur with this technique as even if we open an existential, we must still step the new configuration. Consequently, the problematic composition above would be detected via stepping.

We can use bounded quantification to provide more expressive abstractions that go beyond the fully opaque types used above (which are equivalent to a "$<:$ **top**" bound), and convert this example into one of more practical use. By using appropriate bounds, we can give concrete roles to the `Nothing` and `Full` protocols. Consider that we want to share access to some data structure among several different threads. However, depending on how these threads dynamically use that data structure, it may become important to switch its representation (such as change from a linked list to a binary tree, etc.). Furthermore, we want one specialized thread (the C*ontroller*) to retain precise control over the data structure and to be allowed to monitor and change its representation. Concurrently, an arbitrary number of other threads (the W*orkers*) also have access to the data structure but are limited to only access its Basic operations.

$$\begin{array}{lcl}\mathtt{W} & \triangleq & \exists X <: \mathtt{B}.(\ \mathbf{rw}\ p\ X \Rightarrow \mathbf{rw}\ p\ X\ ;\ \mathtt{W}\ ) \\ \mathtt{C}[Y] & \triangleq & \mathbf{rw}\ p\ Y \Rightarrow \forall Z <: \mathtt{B}.(\ \mathbf{rw}\ p\ Z\ ;\ \mathtt{C}[Z]\ )\end{array}$$

As before `W` is committed to preserve the representation type of X although it now has sufficient room to use that type as `B`. `C` is now more constrained than before since it is forced to guarantee a type that is compatible with `B`. However, `C` retains the possibility of both changing the representation type contained in the shared state, and also of "remembering" the precise (representation) type that was the result of its own local action. Finally, note that we can safely re-split `W` arbitrarily (i.e. `W` $\Rrightarrow$ `W` $||$ `W`). Protocol composition yields similar set of configurations, but with the bound assumption on Z. This form of asymmetric interaction over shared state relates to the `full` $-$ `pure` interaction of *access permissions* [3].

A `full` permission allows exclusive write permission to an object, but also enables read-only permissions (`pure`) to co-exists. Consequently, each `pure` permission must assume that other permissions can modify the shared object up to a certain type, the *state guarantee*. While their work focuses on the read-write distinction, and our work is centered on modeling type-changing mutations (so all aliases can write), the example shows that we are able to naturally model similar asymmetric interaction within our protocol framework.

3.2 Inner Step Extension with Specialization

Re-splitting an existing protocol while specializing its interference is possible, provided that its effects remain consistent with those of the original protocol. Namely we can append new steps to an otherwise ownership recovery step, or produce effects that are more precise than those of the original protocol. The first case allows us to connect two protocols together by that recovery step. The latter case is more interesting: when combined with abstraction it allows specialization *within* an existing step (i.e. nested re-splits), enabling new forms of shared state interactions through that abstraction.

To illustrate the expressiveness gains, we revisit the join protocol of Section 2. However, instead of spawning a single thread to compute the work, we re-split the join protocol in two symmetric workers that share the workload. The last of the workers to complete merges the two results together and "signals" the waiting main thread. First, we rewrite the two protocols to enable abstraction on the `M` protocol, and add a choice (&) to the `F` protocol that enables `F` to use the state more than once until it provides a result.

$$\texttt{F}[X] \triangleq (\ \textbf{rw}\ p\ \texttt{W\#}X \Rightarrow \forall Y.(\ \textbf{rw}\ p\ \texttt{W\#}Y\ ;\ \texttt{F}[Y]\ )\ )\ \&\ (\ \textbf{rw}\ p\ \texttt{W\#}X \Rightarrow \textbf{rw}\ p\ \texttt{R\#}\textbf{int}\ ;\ \textbf{none}\ )$$
$$\texttt{M} \triangleq \exists Z.(\ \textbf{rw}\ p\ \texttt{W\#}Z \Rightarrow \textbf{rw}\ p\ \texttt{W\#}Z\ ;\ \texttt{M}\ )\ \oplus (\ \textbf{rw}\ p\ \texttt{R\#}\textbf{int} \Rightarrow \textbf{rw}\ p\ \textbf{int}\ ;\ \textbf{rw}\ p\ \textbf{int}\ )$$

As before, `M` will `Wait` until there is a `Result` in p. At that point, `M` will recover ownership of that cell. Unlike before, `M` no longer depends on the value tagged as `W` since it is abstracted as Z. The `F` protocol now holds two choices (&): the old step that transitions from `Wait` to `Result`, and a new step that changes the representation of the value tagged as `W` and used during the wait phase. The `F` protocol of Section 2 is a specialization of this protocol since it includes only one of the choices. In here, we specialize `F` into two symmetric worKer protocols. To simplify the presentation, we assume that the worker thread will receive the work parameters through some other mean (such as a pure value shared among threads). Once a worker finishes its job, it will push the resulting `int` to the shared state. If it notices it is the last worker to finish, it will merge the two results together and flag the state as ready, so that `Main` can proceed.

$$\texttt{K} \triangleq (\ \textbf{rw}\ p\ \texttt{W\#}(\texttt{E\#}[]) \Rightarrow \textbf{rw}\ p\ \texttt{W\#}(\texttt{R\#}\textbf{int})\ ;\ \textbf{none}\ )\ \oplus\ (\ \textbf{rw}\ p\ \texttt{W\#}(\texttt{R\#}\textbf{int}) \Rightarrow \textbf{rw}\ p\ \texttt{R\#}\textbf{int}\ ;\ \textbf{none}\ )$$

It is important to note that the new tags/values are nested *inside* the old `W` tag. This ensures that the new usages remain hidden from `M` and "look" just like the previous `F` usage. (There are also no lifetime conflicts since `M` does not preserve its type assumption on the abstraction beyond a single step.) However, these inner tags are used by the two workers for coordination: the `W#Empty` tag means that neither thread has finished, and `W#Result` means that one of the threads has already finished. We can then re-split `F` as follows (note the required initial type in `F`, `E#[]`, for this split to be valid): $\Gamma \vdash \texttt{F}[\texttt{E\#}[]] \Rrightarrow \texttt{K} \parallel \texttt{K}$. Protocol composition follows analogous principles to above, except that we are now simulating the steps of the original `F` protocol with the steps of the two new `K` protocols:

$$\{\ \langle\ \Gamma \vdash \texttt{F}[\texttt{E\#}[]] \Rrightarrow \texttt{K} \parallel \texttt{K}\ \rangle\ ,\ \langle\ \Gamma \vdash \texttt{F}[\texttt{R\#}\textbf{int}] \Rrightarrow \textbf{none} \parallel \texttt{K}\ \rangle\ ,\ \langle\ \Gamma \vdash \texttt{F}[\texttt{R\#}\textbf{int}] \Rrightarrow \textbf{none} \parallel \textbf{none}\ \rangle\ ,\ \langle\ \Gamma \vdash \textbf{none} \Rrightarrow \textbf{none} \parallel \textbf{none}\ \rangle\ \}$$

$$\frac{\langle\ \Gamma_0 \vdash R \Rrightarrow \mathcal{R}[P]\ \rangle \mapsto C}{\langle\ \Gamma_0, \Gamma_1 \vdash R \Rrightarrow \mathcal{R}[P]\ \rangle \mapsto C}\ (\text{C-RS:WEAKENING}) \qquad \frac{\langle\ \Gamma, l : \mathbf{loc} \vdash S \Rrightarrow \mathcal{R}[S \Rightarrow P]\ \rangle \mapsto C}{\langle\ \Gamma \vdash S \Rrightarrow \mathcal{R}[S \Rightarrow \forall l.P]\ \rangle \mapsto C}\ (\text{C-SS:FORALLLOC})$$

$$\frac{\langle\ \Gamma \vdash S \Rrightarrow \mathcal{R}[P\{p/l\}]\ \rangle \mapsto C}{\langle\ \Gamma \vdash S \Rrightarrow \mathcal{R}[\exists l.P]\ \rangle \mapsto C}\ (\text{C-SS:OPENLOC}) \qquad \frac{\langle\ \Gamma, X : \mathbf{type}, X <: A \vdash S \Rrightarrow \mathcal{R}[S \Rightarrow P]\ \rangle \mapsto C}{\langle\ \Gamma \vdash S \Rrightarrow \mathcal{R}[S \Rightarrow \forall X <: A.P]\ \rangle \mapsto C}\ (\text{C-SS:FORALLTYPE})$$

$$\frac{\Gamma \vdash A_1 <: A_0 \qquad \langle\ \Gamma \vdash S \Rrightarrow \mathcal{R}[P\{A_1/X\}]\ \rangle \mapsto C}{\langle\ \Gamma \vdash S \Rrightarrow \mathcal{R}[\exists X <: A_0.P]\ \rangle \mapsto C}\ (\text{C-SS:OPENTYPE})$$

$$\frac{\langle\ \Gamma, X : \mathbf{type}, X <: A \vdash P \Rrightarrow \mathcal{R}[Q]\ \rangle \mapsto C}{\langle\ \Gamma \vdash \exists X <: A.P \Rrightarrow \mathcal{R}[\exists X <: A.Q]\ \rangle \mapsto C}\ (\text{C-PS:EXISTSTYPE}) \qquad \frac{\langle\ \Gamma, l : \mathbf{loc} \vdash P \Rrightarrow \mathcal{R}[Q]\ \rangle \mapsto C}{\langle\ \Gamma \vdash \exists l.P \Rrightarrow \mathcal{R}[\exists l.Q]\ \rangle \mapsto C}\ (\text{C-PS:EXISTSLOC})$$

$$\frac{\langle\ \Gamma, X : \mathbf{type}, X <: A \vdash S \Rightarrow P \Rrightarrow \mathcal{R}[S \Rightarrow Q]\ \rangle \mapsto C}{\langle\ \Gamma \vdash S \Rightarrow \forall X <: A.P \Rrightarrow \mathcal{R}[S \Rightarrow \forall X <: A.Q]\ \rangle \mapsto C}\ (\text{C-PS:FORALLTYPE})$$

$$\frac{\langle\ \Gamma, l : \mathbf{loc} \vdash S \Rightarrow P \Rrightarrow \mathcal{R}[S \Rightarrow Q]\ \rangle \mapsto C}{\langle\ \Gamma \vdash S \Rightarrow \forall l.P \Rrightarrow \mathcal{R}[S \Rightarrow \forall l.Q]\ \rangle \mapsto C}\ (\text{C-PS:FORALLLOC}) \qquad \frac{\langle\ \Gamma \vdash S \Rightarrow P\{p/l\} \Rrightarrow \mathcal{R}[S \Rightarrow Q]\ \rangle \mapsto C}{\langle\ \Gamma \vdash S \Rightarrow \forall l.P \Rrightarrow \mathcal{R}[S \Rightarrow Q]\ \rangle \mapsto C}\ (\text{C-PS:LOCAPP})$$

$$\frac{\Gamma \vdash A_1 <: A_0 \qquad \langle\ \Gamma \vdash S \Rightarrow P\{A_1/X\} \Rrightarrow \mathcal{R}[S \Rightarrow Q]\ \rangle \mapsto C}{\langle\ \Gamma \vdash S \Rightarrow \forall X <: A_0.P \Rrightarrow \mathcal{R}[S \Rightarrow Q]\ \rangle \mapsto C}\ (\text{C-PS:TYPEAPP})$$

$P\{A/X\} \triangleq$ "substitution, in P, of X for A"
Note: bound type/location variables of a type must be fresh in that rule's conclusion.

Figure 7 Protocol composition abstraction extension.

Each simulation will match the rely and guarantee types of a step in `F` with a step in `K`, even if specializing a $\forall$ of `F` to a specific type in `K`. As before, `K` can choose which step to simulate when given a choice (&) of `F` steps. Similarly, at least one alternative ($\oplus$) of `K` must match a step in `F`. Therefore, the new `K` protocols work within the interference of the original `F` protocol, but specialize its uses of the shared state.

3.3 Composing Abstract Protocols

The composition rules of Figure 7 complement those of Figure 6 to enable composing abstract protocols. Weakening on a configuration (up to renaming), (C-RS:WEAKENING), is the crucial mechanism that enables us to close the co-inductive proof when using quantifiers. Thus, when we reach a configuration that is equivalent up to renaming of variables and weakening of Γ, we can close the proof. The (C-SS:FORALL*) rules do similar stepping to (C-SS:STEP) but considering an abstracted guarantee, which results in a typing environment with the opened abstraction. (C-SS:OPEN*) exposes the representation type/location (if it exists) before doing a regular step. (C-PS:FORALL*) and (C-PS:EXISTS*) open their respective abstraction before doing a regular simulation step. More interestingly, (C-PS:*APP) enables a simulated step to pick a particular type/location to apply before that regular simulation stepping, enabling step specialization during simulation. In the T.R, we also consider a straightforward extension to protocol composition that enables subtyping over stepping.

```
let newMVar = λ _.
  let m = new Empty#{} in
    // splits the new cell using single MVar protocol
    Γ ⊢ (rw l Empty#[]) ⇛ MVar[l] || none
    {
    putMVar = λ val. rec R.
        lock m;
        case !m of
          Empty#x → m := Full#val;
                    unlock m
        | Full#value → m := Full#value;
                        unlock m;
                        R // retries
        end
      end,
  splitMVar = λ _.
    Γ ⊢ MVar[l] ⇛ MVar[l] || MVar[l]
  {},
  takeMVar = λ _. rec R.
    lock m;
    case !m of
      Empty#x → m := Empty#x;
                 unlock m;
                 R // retries
    | Full#value → m := Empty#{};
                    unlock m;
                    value
    end
  end
```

Figure 8 MVar example.

3.4 Discussion & Brief Examples

Above, we showed how our local, isolated protocol types can model core interference concepts over a relatively small and simple calculus. We refrained from adding support for more precise states and refined data abstractions of others (such as [15]), and focus instead on typestates [20, 30, 29]. However, this is not an intrinsic limitation of our model. If we consider more precise states, we can (for instance) model monotonic counters from prior work [25, 11] where each counter shares state symmetrically. Our local protocols model these uses solely from the perspective of a single alias as:

$$\texttt{MC} \triangleq \exists \underbrace{\{j : \textbf{int}\}}_{J}.(\ \textbf{rw}\ p\ \underbrace{j}_{J} \Rightarrow \forall \underbrace{\{i : \textbf{int} \mid i \geq j\}}_{I}.(\ \textbf{rw}\ p\ \underbrace{i}_{I}\ ;\ \texttt{MC}\ )\ )$$

The protocol models a monotonically increasing counter on location p. The step relies on location p initially containing some integer, j, and modifying the cell to store some other value, i, that is greater or equal than j. This interaction can be reduced to the core existential-universal protocol interaction discussed above (but, in our calculus, using less precise types: J and I) and where the protocol can be re-split indefinitely.

While our states are less precise, we can enforce more precise uses of that shared state. The semantics of prior work [25, 11] differ on whether the counter is forcefully used by clients, or whether the action was simply available to be used. We can model the two cases explicitly: $\exists p.(\ (!\textbf{ref}\ p) :: \texttt{MC} \multimap [] :: \texttt{MC}\ )$, which enables clients to use the counter an arbitrary number of times or simply thread it through, unused. While in:

$$\forall X.\exists p.(\ (!\textbf{ref}\ p) :: \exists J.(\ \textbf{rw}\ p\ J \Rightarrow \forall I.(\ \textbf{rw}\ p\ I\ ;\ X\ )\ ) \multimap [] :: X\ )$$

by unfolding the protocol, the function guarantees that a single step of the protocol will be used. Since we (intentionally) abstract subsequent steps, the function cannot use the counter beyond that single use. Analogous reasoning can be used to enforce specific, finite, usages.

Adding support for dependent refinement types, and ensuring its decidability (even without interference), is beyond the scope of our work as we focus on the core composition problem. However, we believe that the underlying decidability insights made here will carry to a system with decidable dependent refinement types; even if perhaps requiring more fine-grained conditions to close the co-inductive proof of safe interference—that are only relevant once more precise typing is considered.

While we use a relatively simple calculus to keep the theory focused on the core of interference-control, we can for instance model `MVars` [24]. Figure 8 shows an `MVar`, a

```
let x = new 0 in
 // share 'x' via protocols
 {
   lockMe = λ_.lock x,
   // ...
 }
```

Figure 9 Indirect locking.

```
lock @a;  Γ = a : ref @a
  let b = !a;
  // locks location of 'b'
  lock @b;
unlock @a;
    let c = !b;
    lock @c;
  unlock @b;
    ...
```

Figure 10 Hand-over-hand locking example.

structure that contains a single shared cell which is either empty or contains a value of some type. Notable operations include: `putMVar`, that waits until the cell is empty before inserting the given value; and `takeMVar` which waits until the cell is full to remove the cell's value, leaving the cell empty. `MVar`s can be shared by many aliases using the protocol:

$$\begin{aligned}\texttt{MVar}[m] \triangleq\ & \exists Y.(\ (\ (\mathbf{rw}\ m\ \texttt{Empty}\#Y) \Rightarrow (\mathbf{rw}\ m\ \texttt{Empty}\#Y)\ ;\ \texttt{MVar}[m]\ )\ \&\\ & \qquad (\ (\mathbf{rw}\ m\ \texttt{Empty}\#Y) \Rightarrow (\mathbf{rw}\ m\ \texttt{Full}\#\mathbf{int})\ ;\ \texttt{MVar}[m]\ )\ )\\ & \oplus\ (\ (\ (\mathbf{rw}\ m\ \texttt{Full}\#\mathbf{int}) \Rightarrow (\mathbf{rw}\ m\ \texttt{Empty}\#[])\ ;\ \texttt{MVar}[m]\ )\ \&\\ & \qquad \exists Y.(\ (\mathbf{rw}\ m\ \texttt{Full}\#Y) \Rightarrow (\mathbf{rw}\ m\ \texttt{Full}\#Y)\ ;\ \texttt{MVar}[m]\ )\ )\end{aligned}$$

The T.R. [21] includes additional examples, including modeling examples of prior work with our more local protocol types. We can also model a shared pair where each alias keeps its own, local, precise knowledge on one of the two components of the pair stored in that shared state. The two aliases, `L` and `R`, share a common cell but keep part of that state private to itself. While both can do private actions over the shared cell, they are guaranteed to not interfere with the precise assumptions of the remaining alias.

$$\begin{aligned}&\texttt{P}[A][B] \triangleq \mathbf{rw}\ p\ [A,B]\\ &\texttt{L}[A] \triangleq \exists X.(\ \texttt{P}[A][X] \Rightarrow \forall Y.(\ \texttt{P}[Y][X]\ ;\ \texttt{L}[Y]\ )\ )\\ &\texttt{R}[A] \triangleq \exists X.(\ \texttt{P}[X][A] \Rightarrow \forall Y.(\ \texttt{P}[X][Y]\ ;\ \texttt{R}[Y]\ )\ )\end{aligned} \qquad \Gamma \vdash \texttt{P}[X][Y] \Rrightarrow \texttt{L}[X] \parallel \texttt{R}[Y]$$

Thus, we can use the different perspectives of each protocol to model local knowledge that is hidden from other aliases, within our core protocol framework.

Since our types express sharing, we can use standard techniques to abstract the components of a protocol type after safe composition is checked. This enables an abstraction to expose a type interface that indirectly manipulates the shared state, such as indirectly locking/unlocking state (Figure 9). We can type the record in such a way to hide the type in `x` but still expose some information on sharing that is useful for later enabling other typestate functions [20]. For instance: $\exists A.\exists B.\exists C.[\ \ldots,$ `lockMe` $:[] :: (A \Rightarrow B;C) \multimap [] :: (A * (B;C))$, `add` $:(\ \mathbf{int} :: A \multimap [] :: A\ ),\ \ldots\ ]$. Clients can only call `add` once the type A is available. This could model, for instance, a global lock on a collection to enable more coarse-grained control over the interference to that collection—but without exposing the lock to clients. Thus, when `lockMe` returns, the client receives a type that expresses that A is available and that a guarantee $(B;C)$ is expected to be fulfilled. However, this fulfillment can

only occur indirectly via the wrapper record as clients do not have a direct way of accessing or mutating the internals of that shared state.

While we do not guarantee dead-lock freedom, it is possible to type more fine-grained locking schemes such as *hand-over-hand* locking (Figure 10). Consider the protocol of a list's node:

$$\mathsf{L}[q] \triangleq \exists l.(\ (\mathbf{rw}\ q\ !\mathbf{ref}\ l) * \mathsf{L}[l] \quad \Rightarrow \quad (\mathbf{rw}\ q\ !\mathbf{ref}\ l)\ ;\ \ldots\ )$$

L is defined over a location q that contains the (abstracted) reference to the next element of the sequence of $l$ocations to be locked. Locking will enable access to that **ref** l which can then be locked to gain access to $\mathsf{L}[l]$, the next element in the sequence of locations to lock. For brevity, we make each step simply consume L, instead of (for instance) re-splitting.

4 Composition Decidability & Other Technical Results

We now show decidability of protocol composition and discuss the remaining technical results of our language. The decidability statement comes as a direct consequence of ensuring a regular type structure via syntactic well-formedness constraints on recursive types. Although applied in the context of protocol composition, we follow ideas from prior work on ensuring decidable subtyping over bounded quantification [28, 4]. The main novelty is in extending this kind of reasoning to account for recursive types with parameters, in order to ensure a regular type structure over our more flexible recursive types. To achieve this, we apply well-formedness conditions which ensure that there is only a finite number of reachable (abstract) protocol states. We focus the discussion on decidability of protocol composition, and point interested readers to T.R. where these conditions are properly motivated and discussed. Crucially, these well-formedness conditions enable us to state the following:

▶ **Lemma 1** (Finite Uses). *Given a well-formed recursive type* $(\textbf{\textit{rec}}\ X(\overline{u}).A)[\overline{U}]$ *the number of possible uses of* X *in* A *such that* $\Gamma \vdash X[\overline{U'}]$ ***type*** *is bounded.*

▶ **Lemma 2** (Finite Unfolds). *Unfolding a well-formed recursive type* $(\textbf{\textit{rec}}\ X(\overline{u}).A)[\overline{U}]$ *produces a finite set of variants of that original recursive type that (at most) contains: permutations of* $\overline{U}$*, or a set of mixtures of* $\overline{U}$ *with some type/location variables representing a class of equivalent* $(\equiv)$ *types.*

▶ **Lemma 3** (Finite Sub-Terms). *Given a well-formed type* A*, such that* $\Gamma \vdash A$ ***type****, the set of sub-terms of* A *is finite up to renaming of variables and weakening of* Γ*.*

4.1 Composition Properties, Algorithm, and Decidability

Informally, correctness of protocol composition is based on the two properties: 1) a split results in protocols that can always take a step with the current state of the shared resources, thus are never stuck; and, 2) protocol composition is a partial commutative monoid (associative, commutative, and with **none** as the identity element). Because of property 2), iterative splittings of existing protocols remain struck-free, unable to cause unsafe interference. We now state these properties formally but leave the proofs to the T.R.. The next two lemmas show stuck freedom by properties that resemble progress and preservation but over protocols:

▶ **Lemma 4.** *If* $\Gamma \vdash R \Rrightarrow P \mid\mid Q$ *then* $\langle\ \Gamma \vdash R \Rrightarrow P \mid\mid Q\ \rangle \mapsto C$*.*

Meaning that if two protocols, P and Q, compose safely then their configuration can take a step to another set of configurations, C.

▶ **Lemma 5.** *If* $\langle\ \Gamma \vdash R \Rrightarrow P \mid\mid Q\ \rangle \mapsto \langle\ \Gamma' \vdash R' \Rrightarrow P' \mid\mid Q'\ \rangle \cdot\ C$ *and* $\Gamma \vdash R \Rrightarrow P \mid\mid Q$ *then* $\Gamma' \vdash R' \Rrightarrow P' \mid\mid Q'$.

The lemma ensures that if two protocols compose safely, then any of the next configurations that result from stepping will also be safe. Note that protocol composition does not enforce that the shared resources are not lost. Instead our concern is on safe interference. Indeed, resources that are never used will never be able to unsafely interfere. To avoid losing resources, we must forbid the use of (C-RS:NONE) on non-terminated protocols and that both P and Q cannot have both simultaneously terminated if there are non-**none** resources left. Once that restriction is considered, our splitting induces a monoid in the sense that for any P and Q for which $\Gamma \vdash R \Rrightarrow P \mid\mid Q$ is defined there is a single such R (defined up to subtyping and equivalent protocol/state interference specification). Since for any two protocols there may not always exist an R that can be split into P and Q, this is a partial monoid.

▶ **Lemma 6.** *Protocol composition obeys the following properties:*

1. *(identity)* $\Gamma \vdash R \Rrightarrow R \mid\mid$ ***none***.
2. *(commutativity) If* $\Gamma \vdash R \Rrightarrow P_0 \mid\mid P_1$ *then* $\Gamma \vdash R \Rrightarrow P_1 \mid\mid P_0$.
3. *(associativity) If we have* $\Gamma \vdash R \Rrightarrow P_0 \mid\mid P$ *and* $\Gamma \vdash P \Rrightarrow P_1 \mid\mid P_2$ *then exists* Q *such that* $\Gamma \vdash R \Rrightarrow Q \mid\mid P_2$ *and* $\Gamma \vdash Q \Rrightarrow P_0 \mid\mid P_1$.
   (i.e. If $\Gamma \vdash R \Rrightarrow P_0 \mid\mid (P_1 \mid\mid P_2)$ *then* $\Gamma \vdash R \Rrightarrow (P_0 \mid\mid P_1) \mid\mid P_2$ *)*

Protocol composition is defined as a "split", left-to-right ($\Rrightarrow$). Simply reading the rules as right-to-left ($\Lleftarrow$) to compute a "merge" is not safe. For instance, it would enable merging to arbitrary choices with (C-RS:STATEINTERSECTION). Intuitively, merging needs to intertwine the uses of both protocols. However, since we do not track copies (as we target sharing when that tracking is not possible), merging cannot "collapse" a protocol into a non-protocol type. In this case "merging" is equivalent to simply having the two non-merged protocols available in Δ or bundled using the $*$ type.

The composition algorithm is shown in the T.R. and is a straightforward implementation of the axiomatic definitions shown above. The algorithm uses a set of *v*isited configurations to remember past configurations and ensure that once all different protocol configurations are exhausted (up to renaming and weakening of Γ), the algorithm can terminate. We now state our technical lemmas on the composition algorithm but leave the proofs to the T.R..

▶ **Lemma 7.** *Given well-formed types and environment, we have that:*

1. *(soundness) if* $\mathsf{c}(\Gamma,\ R,\ P,\ Q\ )$ *then* $\Gamma \vdash R \Rrightarrow P \mid\mid Q$.
2. *(completeness) if* $\Gamma \vdash R \Rrightarrow P \mid\mid Q$ *then* $\mathsf{c}(\Gamma,\ R,\ P,\ Q\ )$.
3. *(decidability)* $\mathsf{c}(\Gamma,\ R,\ P,\ Q\ )$ *terminates.*

4.2 Correctness Properties

Progress and preservation theorems are defined over valid program configurations such that:

$$\frac{\Gamma \mid \Delta_i \vdash e_i : ![] \dashv \cdot \qquad i \in \{0, ..., n\} \qquad n \geq 0}{\Gamma \mid \Delta_0,\ ...,\ \Delta_n \vdash e_0\ \cdot\ ...\ \cdot\ e_n}\ (\text{WF:PROGRAM})$$

Stating that a thread pool ($e_0\ \cdot\ ...\ \cdot\ e_n$) is well formed if each thread can be assigned a "piece" of the linear typing environment (containing resources), and if each individual expression has type ![] without leaving any residual resources ($\cdot$). Note that the conditions on each thread (e_i) are identical to those imposed by (T:FORK). For clarity, both safety theorems are supported by auxiliary theorems over a single expression, besides the main theorem over the complete thread pool. We now state progress over programs:

▶ **Theorem 8.** *If* $\Gamma \mid \Delta \vdash T_0$ *and* $\texttt{live}(T_0)$ *and if exists* H_0 *such that* $\Gamma \mid \Delta \vdash H_0$ *then* $H_0 \; ; \; T_0 \; \mapsto \; H_1 \; ; \; T_1$.

$\texttt{live}(T)$ means that the thread pool T contains at least one "live" thread such that the thread is neither a value nor is waiting for a lock to be released (which includes deadlocks). $\Gamma \mid \Delta \vdash H$ ensures that the Heap is well-defined according to Γ and Δ.

We define $\texttt{Wait}(H, e)$ over a thread e and heap H such that the $\mathcal{E}$valuation context is reduced to evaluating the configuration: $H \; ; \; \mathcal{E}[\mathsf{lock}\ \rho, \overline{\rho'}] \; \cdot \; T$ where $\rho \hookrightarrow v \notin H$ which contains at least one location (ρ) that is currently locked or was deleted and, therefore, the thread must block waiting (potentially indefinitely) for that lock to be available before continuing. "Early" deletion of shared resources results in a pending guarantee. Since well-formed threads cannot leave residual resources, this situation is ruled out for correct programs, but may occur on the theorem below. Progress over expressions is defined as:

▶ **Theorem 9.** *If* $\Gamma \mid \Delta_0 \vdash e_0 : A \dashv \Delta_1$ *then we have that either:*

- e_0 *is a value, or;*
- *if exists* H_0 *and* Δ *such that* $\Gamma \mid \Delta, \Delta_0 \vdash H_0$ *then either:*
  - *(steps)* $H_0 \; ; \; e_0 \; \mapsto \; H_1 \; ; \; e_1 \; \cdot \; T$, *or;*
  - *(waits)* $\texttt{Wait}(H_0, e_0)$.

Preservation ensures that a reduction step will preserve both the type and the effects of the expression that is being reduced (so that each thread's type, ![], and effect, ·, remains unchanged). As above, we use a preservation theorem over programs that makes use of an auxiliary theorem on preservation over expressions:

▶ **Theorem 10.** *If we have* $\Gamma_0 \mid \Delta_0 \vdash H_0$ *and* $\Gamma_0 \mid \Delta_0 \vdash T_0$ *and* $H_0 \; ; \; T_0 \; \mapsto \; H_1 \; ; \; T_1$ *then, for some* Δ_1 *and* Γ_1, *we have:* $\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash H_1$ *and* $\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash T_1$.

So that a well-formed pool of threads (T_0) remains well-formed after stepping one of these threads (resulting in T_1). Preservation over a single expression must still account for the resources (Δ_T) that may be consumed by a newly spawned thread (T):

▶ **Theorem 11.** *If we have* $H_0 \; ; \; e_0 \; \mapsto \; H_1 \; ; \; e_1 \; \cdot \; T$ *and* $\Gamma_0 \mid \Delta_0, \Delta_T, \Delta_2 \vdash H_0$ *and* $\Gamma_0 \mid \Delta_0, \Delta_T \vdash e_0 : A \dashv \Delta$ *then, for some* Δ_1 *and* Γ_1, *we have:* $\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_T, \Delta_2 \vdash H_1$ *and* $\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A \dashv \Delta$ *and* $\Gamma_0, \Gamma_1 \mid \Delta_T \vdash T$.

The following "Error Freedom" corollary complements the main results to show that our system cannot type programs that allow data races and the dereference of destroyed memory cells, i.e. that our system ensures memory safety and race freedom.

▶ **Corollary 12.** *The following program states cannot be typed:*

1. *(Data Race)* $H; \mathcal{E}_0[\rho := v] \cdot \mathcal{E}_1[!\rho] \cdot T \qquad H; \mathcal{E}_0[\rho := v] \cdot \mathcal{E}_1[\rho := v'] \cdot T$
2. *(Memory Fault)* $H; \mathcal{E}[\rho := v] \cdot T \qquad H; \mathcal{E}[!\rho] \cdot T \qquad$ *(where* $\rho \notin H$*)*
3. *(Ownership Fault)* $H; \mathcal{E}[\mathsf{delete}\ \rho] \cdot T \qquad$ *(where* $\rho \notin H$*)*

The proof is straightforward due to our use of locks to ensure mutual exclusion and the fact that our protocols discipline the use of shared state. Thus, these errors are ruled out by either protocol composition or by the resource tracking of the core linear system.

```
receive(c) ≜ rec R.
    lock c;
    case !c of
      // 1. waiting states (A..Z)
      A#n → ... // analogous to below
    | Z#n → // restore linear content
        c := Z#n;
        unlock c; R // retry
      // 2. desired (receive) state
    | ReadyToReceive#v →
        c := Idle#{}; // "received"
        unlock c;
        v // value received from "channel"
    end
  end
```

```
send(c,v) ≜ rec R.
    lock c;
    case !c of
      // 1. waiting states (A..Z)
      A#n → ... // analogous to below
    | Z#n → // restore linear content
        c := Z#n;
        unlock c; R // retry
      // 2. desired (idle) state
    | Idle#_ →
        c := ReadyToReceive#v; // "sent"
        unlock c;
        {} // result of send is empty
    end
  end
```

Figure 11 Simple encoding of `send` and `receive` functions via a shared cell.

```
let c = connectSeller() in
c : buy!(prod) ; price?(p) ; details?(d)
  send(c, GET_USER_PRODUCT() );
  let price = receive(c) in
    let details = receive(c) in
      close(c)
    end
  end
```

Figure 12 Buyer code.

5 Protocol Expressiveness

We show the expressiveness of our protocols by modeling typeful message-passing concurrency, using a straightforward encoding of message-passing via shared memory interference (Figure 11). The encoding itself should be unsurprising as it follows well-known ideas from the literature, so we defer less important details to the T.R. to focus instead on the most interesting aspect of this example: how our protocol framework is able to type such uses and ensure their safety.

We encode a more primitive, "low-level" view of typeful message-passing concurrency via the causality of shared memory interference. We focus on the non-distributed setting where a channel can be precisely encoded as a low-level shared cell. Channel communication and its changing session properties are emulated indirectly via inspection of or interference over the contents of that shared cell. Thus, our functions to send/receive a value simply hide the underlying Waiting states, for instance to receive:

$$\mathsf{Wait}[A_0..A_n] \oplus \ (\ \mathtt{rw\ c\ ReadyToReceive}\#V \Rightarrow \mathtt{rw\ c\ Idle}\#[]\ ;\ \mathtt{NextStep}\ )$$

where Wait is a sequence of retry steps that leave the state unmodified, until a value of type V is "received". Sending uses a similar protocol but where we must wait for an `Idle` cell before "sending". The T.R. includes the complete "`Buyer-Seller-Shipper`" example (the canonical and simple example used in session-based concurrency works) while in here we only take a look at the main aspects of the `Buyer`'s interaction with the channel (Figure 12).

We model a *c*hannel using a capability to location c. For brevity, we omit "**rw** c" from "**rw** c A" since all changes occur over that same location. The `Buyer`'s type uses standard π-calculus [22] notations where ! sends and ? receives a value. These actions are mapped to the rely type (receive) and the guarantee type (send).

$$\underbrace{\texttt{buy}!(prod)}_{\texttt{idle0}\#[] \Rightarrow \texttt{buy}\#prod} \;;\; \underbrace{\texttt{price}?(p)}_{\texttt{price}\#p \Rightarrow \texttt{idle2}\#[]} \;;\; \underbrace{\texttt{details}?(d)}_{\texttt{details}\#d \Rightarrow []}$$

`Buyer` starts by sending a request to `buy` some *prod*uct, then waits for the *p*rice, and finally receives the *d*etails of that product. Under that interaction protocol, we simply map sends to a guarantee type of a step, and receives to a rely type of a step.

Our protocol interactions are both non-deterministic and may contain an arbitrary number of simultaneous participants. To ensure that the desired participant (`Buyer`) is the only one allowed to received (take) the price, we must mark the contents with a specific tag so that only `Buyer` has permission to change that state. To handle the non-deterministic interleaving of protocols, we must introduce explicit "wait states" that allow a participant to check if the communication has reached the desired point to that participant or if it should continue waiting. We abstract these steps as Wait as they simply recur on that same step:

$$\texttt{idle0}\#[] \Rightarrow \texttt{buy}\#prod \;;\; \mathsf{Wait} \oplus (\, \texttt{price}\#p \Rightarrow \texttt{idle2}\#[] \,) \;;\; \mathsf{Wait} \oplus (\, \texttt{details}\#d \Rightarrow [] \,)$$

The richness of our shared state interactions means that we can immediately support fairly complex session-based mechanisms (such as delegation, asynchronous communication, "messages to self", multiparty interactions, internal/external choices, etc.) within our small protocol framework. However, this flexibility comes at the cost of requiring a more complex composition mechanism. Protocol composition accounts for both non-deterministic protocol interleaving and "multi-way" communication, features which are usually absent from strictly choreographed session-based concurrency (favoring instead strong liveness properties over more deterministic, linear compositions). Naturally, more complex examples are possible. In here our focus is on showing the core insights that enable us to relate the two techniques: 1) mapping receive/send to our rely/guarantee types; 2) adding explicit waiting states to account for non-deterministic protocol interleaving; and 3) tag the content of a cell in order to ensure that only the right participant will be able to mutate the state at that point in the interaction. (Recall that we do not guarantee deadlock freedom, nor termination.)

6 Related Work

This work is based on results collected in Militão's PhD thesis [18]. Our work relates to prior work on *rely-guarantee protocols* [19]. We show that these protocols are useful to reason about concurrency and significantly improve the flexibility of protocol composition. Namely, we allow the composition of abstract protocols (enabling more local typing), show that our composition is decidable, and provide a novel axiomatic definition of composition that is straightforward to implement. Since thread-based interference is rooted in alias-related interference, the technique itself is mostly indifferent to whether sharing occurs in the sequential or concurrent setting. Still, we address all technicalities that make concurrency possible, such as adding support for threads and locking of locations; which then enables us to express typeful message-passing concurrency in our protocols.

Our work is also related to recent work on more precise tracking of interference. *Chalice* [17] uses a simplified form of rely-guarantee to reason about shared state interference by constraining a thread's changes to a two-state invariant, relating the previous and current states. Monotonic [8, 25] uses of shared state (where all changes converge to more precise states) are less dependent on aliasing information, which simplifies checking at the expense of expressiveness. Dynamic ownership recovery mechanisms [33, 26] choose some run-time overhead and dynamic safety guarantees to enable more flexible ownership recovery than

purely static approaches. *Rely-guarantee references* [11] adapt the use of rely-guarantee to individual reference cells with support for dependent refinement types in a sequential language. Although the use of refinements adds expressiveness to the description of sharing, they do not support ownership recovery, nor address decidability, and typechecking can require manual assistance in Coq. *Access permissions* [33, 3, 2] control alias interference by categorizing read-write uses into different permission kinds. Our design omits the read-write distinction to focus exclusively on structuring alias interference using more fundamental protocol primitives. Interestingly, although we only model write-exclusive uses, our types can enforce effectively read-exclusive semantics by ensuring that any private change in a cell will be reverted to its original public value. However, this simpler form of read-only cannot capture their multiple, simultaneous readers case. Still, by modeling interference in a more fundamental way, we gain additional expressiveness beyond their most permissive `share` permission as we can model uses beyond invariant-based sharing. In [5] Crafa and Pavodani introduce a high-level (actor-like) model for sharing (type)state via join patterns. We target a more low-level programming paradigm (which builds typestates through type abstraction rather than as a first-class language feature), enabling us to introduce abstraction at the level of protocols and support protocol re-splitting in ways that are not expressible in their work.

Several recent works use *partial commutative monoids* [7, 16, 6] to model sharing by leveraging the concept of fictional separation [7, 12]. Commutative monoids offer the underlying general principle for splitting resources, enabling seemingly unrelated components to interact via aliasing under a layer of (fictional) separation. We compare more closely to [16] due to our common use of $\mathbf{L}^3$ [1] and type-based approach. In [16], Krishnaswami *et al.* define a generic sharing rule based on programmer-supplied commutative monoids for safe sharing of state in a single-threaded environment. Their work does not approach the issue of decidability of resource splitting, and requires wrapping access to shared state in an module abstraction that serves as an intermediary to sharing. Our work focuses on a custom commutative monoid that enables first-class sharing without (necessarily) needing a wrapping module abstraction. Although our protocol splitting is a specialized monoid, we showed that this mechanism is relatively flexible, decidable, and give an algorithmic implementation. Other technical differences between our works abound such as their use of affine refinement types (enabling more fine-grained types), our use of multi-threaded semantics and allowing inconsistent states (i.e. locked cells) to be moved around as first-class, etc.

Protocol-based mechanisms for safe interference are also used by other approaches, such as in program logic-based systems (e.g. [14, 31, 32, 23, 9]). By generally targeting manual proofs (and somewhat more involved specifications) these works generally fit into a different design space than ours, although share some interesting similarities. While we make concessions on expressiveness to achieve decidable protocol composition and re-splitting, these works focus instead on the expressiveness of their concurrency specification. LRG [9] supports lock-free structures but requires a special frame-rule to support framing over rely-guarantee conditions. We simply integrate protocols into the language (as linear resources) meaning that the standard frame-rule suffices. Supporting lock-free concurrency in our system would require reinterpreting a $\Rightarrow$ step as a single-cell `atomic` conditional operation; with the shared resource (stored in the cell) being immediately extracted/inserted from/into the cell, rather than just accessible after locking. CaReSL [32] and Iris [14] support "islands"/regions of memory that are shared together and whose imprecise state must be considered on use. Our composition rules enforce that a protocol carries all information on imprecise states, which is then deconstructed via (T:Alternative-Left) and case analysis. Our protocols can group shared state using the $*$ operator to define shallow "regions", while their works allow

for richer specifications of atomic regions of any depth. Iris [14] further supports a form of re-splitting via a "view shifting" mechanism, to repartition (or create) shared regions. FCSL [23] encodes protocols via auxiliary/ghost state. Although done in a compositional way, it can require checking for safe interference ("stability") after a split since a safe split does not necessarily imply safe interference in all situations. Our composition mechanism is essentially a form of checking for safe interference early, at the moment of the split, by checking that *all* possible future uses are safe (like a form of "pre-computed" stability check).

Protocol composition itself can also be seen as a form of model checking (to check that each state has a successor) that uses abstract states to ensure a finite state space, but in a system that is more intimately integrated with the language. Our protocols are first-class resources that can be specialized by clients, even abstracting (leaving out) later steps. Thus, our protocols guide the programmer on how to reason locally about (safe) interference by mapping its uses of locks to a local protocol type that models the alias perspective on the shared state. While our work focuses on modeling the core interference phenomenon within a small calculus, rather than precisely typing existing programs, we still showed that extensions may be used to model at least some existing programs within our model.

7 Conclusions

We defined a flexible and decidable procedure that ensures the safe composition of interfering abstract protocols that share access to mutable state. While employing a relatively small protocol framework, we are able to model the core interference principles of complex shared state interactions within our core calculus. Finally, we showed the expressiveness of our protocol framework by discussing how it can also model typeful message-passing concurrency.

Acknowledgements. We thank the anonymous reviewers for their helpful feedback.

References

1 A. Ahmed, M. Fluet, and G. Morrisett. L3: A linear language with locations. *Fundam. Inf.*, 77(4):397–449, December 2007.
2 N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *OOPSLA 2008*.
3 K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA 2007*.
4 G. Castagna and B. C. Pierce. Decidable bounded quantification. In *POPL 1994*.
5 S. Crafa and L. Padovani. The chemical approach to typestate-oriented programming. In *OOPSLA 2015*.
6 T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In *POPL 2013*.
7 T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP 2010*.
8 M. Fähndrich and K. Rustan M. Leino. Heap monotonic typestate. In *IWACO 2003*.
9 X. Feng. Local rely-guarantee reasoning. In *POPL '09*.
10 J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.
11 C. S. Gordon, M. D. Ernst, and D. Grossman. Rely-guarantee references for refinement types over aliased mutable data. In *PLDI 2013*.
12 J. B. Jensen and L. Birkedal. Fictional separation logic. In *ESOP 2012*.
13 C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst. 1983*.

14 R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL 2015*.

15 N. R. Krishnaswami, P. Pradic, and N. Benton. Integrating linear and dependent types. In *POPL 2015*.

16 N. R. Krishnaswami, A. Turon, D. Dreyer, and D. Garg. Superficially substructural types. In *ICFP 2012*.

17 K. R. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP 2009*.

18 F. Militão. *Rely-Guarantee Protocols for Safe Interference over Shared Memory*. PhD thesis, Carnegie Mellon University and Universidade Nova de Lisboa, 2015.

19 F. Militão, J. Aldrich, and L. Caires. Rely-guarantee protocols. In *ECOOP 2014*.

20 F. Militão, J. Aldrich, and L. Caires. Substructural typestates. In *PLPV 2014*.

21 F. Militão, J. Aldrich, and L. Caires. Composing interfering abstract protocols (technical report). CMU-CS-16-103, 2016.

22 R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Inf. Comput.*, September 1992.

23 A. Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP 2014*.

24 S. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *POPL 1996*.

25 A. Pilkiewicz and F. Pottier. The essence of monotonic state. In *TLDI 2011*.

26 F. Pottier and J. Protzenko. Programming with permissions in mezzo. In *ICFP 2013*.

27 J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. Logic in Computer Science*, pages 55–74, 2002.

28 J. Seco and L. Caires. Subtyping first-class polymorphic components. In *ESOP 2005*.

29 R. E. Strom. Mechanisms for compile-time enforcement of security. In *POPL 1983*.

30 R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

31 K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *ESOP 2014*.

32 A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ICFP '13*.

33 R. Wolff, R. Garcia, É. Tanter, and J. Aldrich. Gradual typestate. In *ECOOP 2011*.

The Elements of Decision Alignment

Mark S. Miller[1] and Bill Tulloh[2]

1 erights@google.com
2 btulloh@gmail.com

Abstract

When one object makes a request of another, why do we expect that the second object's behavior correctly satisfies the first object's wishes? The need to cope with such *principal-agent problems* shapes programming practice as much as it shapes human organizations and economies. However, the literature about such plan coordination issues among humans is almost disjoint from the literature about these issues among objects. Even the terms used are unrelated.

These fields have much to learn from each other—both from their similarities and from the causes of their differences. We propose a framework for thinking about *decision alignment* as a bridge between these disciplines.

1998 ACM Subject Classification D.2 Software Engineering, F.3 Logics and Meanings of Programs

Keywords and phrases economics, law, contracts, principal-agent problem, incentive alignment, least authority, verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.17

1 Extended Abstract

Many complex systems can be described as networks of principal-agent relationships, in which a requesting entity, the *principal*, sends a request to another entity, the *agent*, who is expected to perform the actions the principal intends. Of course, an agent may decide to do something else instead. Thus, these requests usually occur within a framework in which the principal uses various techniques to try to align the decision of the agents with the interests of the principal.

Economics, organizational theory and political science study principal-agent relationships among humans. Language designers, programmers, and software engineers deal with principal-agent relationships among computational objects. When both principal and agent are humans, the hazards and techniques are examined in terms of divergent interests, asymmetric information, contracts, and incentives[1, 4]. When both principal and agent are objects, the hazards and techniques are examined in terms of abstraction mechanisms, object design patterns, expressiveness, and reliability engineering. In the human case, agents are assumed to have their own interests, and must therefore be coaxed into performing what the principal wants. In the object case, the agent's intentions are generally assumed benign[1], so the goal is to minimize unintentional misbehavior.

A human principal makes requests of a software agent via a user interface. In *HCI*, the study of human-computer interaction, the agent is generally assumed benign and bug-free. The focus is on request expressiveness and user confusion. The hazard is that the agent does

[1] Except computer security, which does consider intentional misbehavior

30th European Conference on Object-Oriented Programming (ECOOP 2016).
Editors: Shriram Krishnamurthi and Benjamin S. Lerner; Article No. 17; pp. 17:1–17:5

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 The Elements of Decision Alignment.

	Human to Human	Human to Object	Object to Object
Explanation	Language	User interface	Abstraction
Rights	Law, Contracts	App permissions Powerbox	Security Protection patterns
"Incentives"	Economics	Objective functions	Machine learning Agorics
Static Inspection	Accounting controls	Trusted path URL bar	Types, Verification Open source eyeballs
Dynamic Monitoring and Feedback	Reviews, Complaints Word of mouth	Bug reports	Contracts, Testing Backprop
Trust and Reputation	Trademark Chain of custody	App stores White and black lists	Trusted developer Same origin

what the user asks for rather than what the user wants. HCI examines how to design an agent's interface to shape user behavior so that users ask for what they want.

Mixed cases are important. Humans increasingly use user-interfaces to express intent and interact with other humans. Indeed, human-to-human intentions are increasingly expressed, constrained, and transformed through software intermediaries, whether email, social networks, or ride sharing.

A unified view, that looks at the different techniques used to deal with each of these three cases, can provide important insights. We identify six broad categories of techniques principals use to align agent decisions with the principal's intent. A common theme throughout is that the principal and agent know different things. By virtue of specialization, they each embody information inaccessible to the other.

Explanation The principal must express its intent as a request that the agent can understand. Only the principal has detailed knowledge of *why* they make this request. Only the agent has detailed knowledge of *how* to fulfill the request. The request conveys *what* the principal needs in terms of concepts they share[5, 10]. A good abstraction boundary composes their separate knowledge to achieve a greater result[2, 8].

Rights All actions that the agent performs, whether it serves the principal's interests or not, must be among the actions that are possible for the agent to take. If the agent's range of possible actions is narrow, this limits its abilities to misbehave in ways that harm the principal. But if too narrow, then the agent cannot do what the principal asks[9]. We return to this issue below.

"Incentives" Some agents change their behavior in order to optimize some externally provided metric. These can be humans responding to prices, or machine learning systems optimizing an objective function. In both cases, the metric might be a proxy measure for a complex composition of goals. By constructing or influencing this metric, principals shape the behavior of such agents to serve goals that the principal cannot otherwise articulate[3].

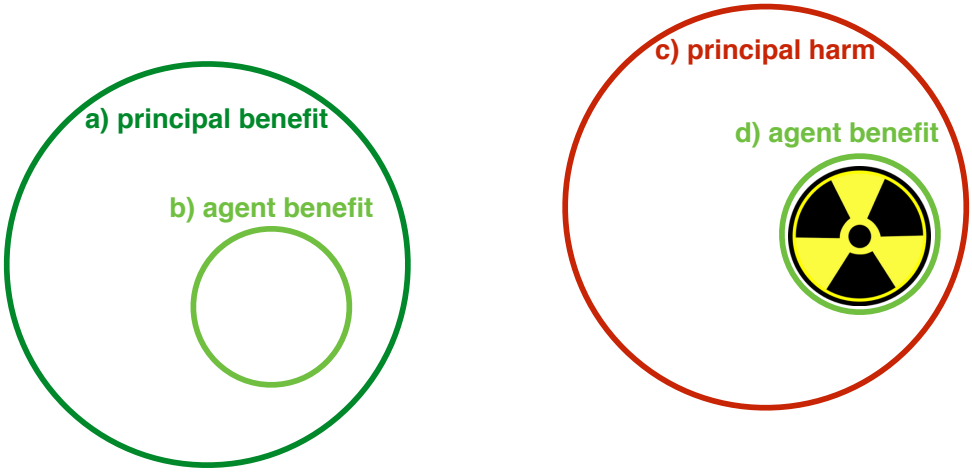


Figure 1 Incentive landscape.

Static Inspection Some agents are internally constructed to be unable to attempt certain behaviors. A business keeps *front-office* and *back-office* separate so that it cannot misbehave in certain ways without a conspiracy of two[2]. Within a statically type-safe language, a principal knows that an agent cannot violate its own interface type. Static verification of complex behavioral constraints is already practical and rapidly improving[7].

Dynamic Monitoring and Feedback The techniques above apply before the agent starts to act. Afterwards, the principal may look at what the agent is actually doing or has done. Depending on its judgement, the principal may change how it continues to employ the agent. The principal may send the judgement as feedback so the agent can learn or be fixed. It may share this feedback with other potential principals.

Trust and Reputation Even with all the safeguards, it can still be risky for the principal to rely on the agent's good behavior. Principals attempt to reduce this risk by putting their trust in the identity of the agent or the agent's origin. Companies trust their own developers. Consumers trust known brands. Feedback from past performance affects reputation, impacting future decisions to take such risks.

While each category provides useful lessons, the real payoff comes from how they interact and reinforce one another. For example, the co-design of an agent's rights and incentives can lead to better results than focusing on each individually.

Figure 1 illustrates a classic principal-agent dilemma. The space represents possible agent behaviors. Circle (a) contains agent behaviors that would benefit the principal. Circle (b) contains agent behaviors that the principal knows how to ask for and reward, benefiting the agent as well. Circle (c) contains agent behaviors that would harm the principal. Circle (d) is the region of greatest hazard, where the principal's harm coincides with the agent's benefit. Often, an incentive-only approach cannot eliminate this area at reasonable cost.

Figure 2 illustrates how practicing *POLA*, the *principle of least authority*[3], can improve the situation. POLA says that, ideally, a principal should grant the agent the narrowest rights that the agent needs to fulfill the principal's request.

[2] Barings learned this the hard way when it gave Nick Leeson both front-office and back-office privileges.
[3] The principle of least authority refines *the principle of least privilege*[9] to clarify what kinds of rights we need to minimize[6].

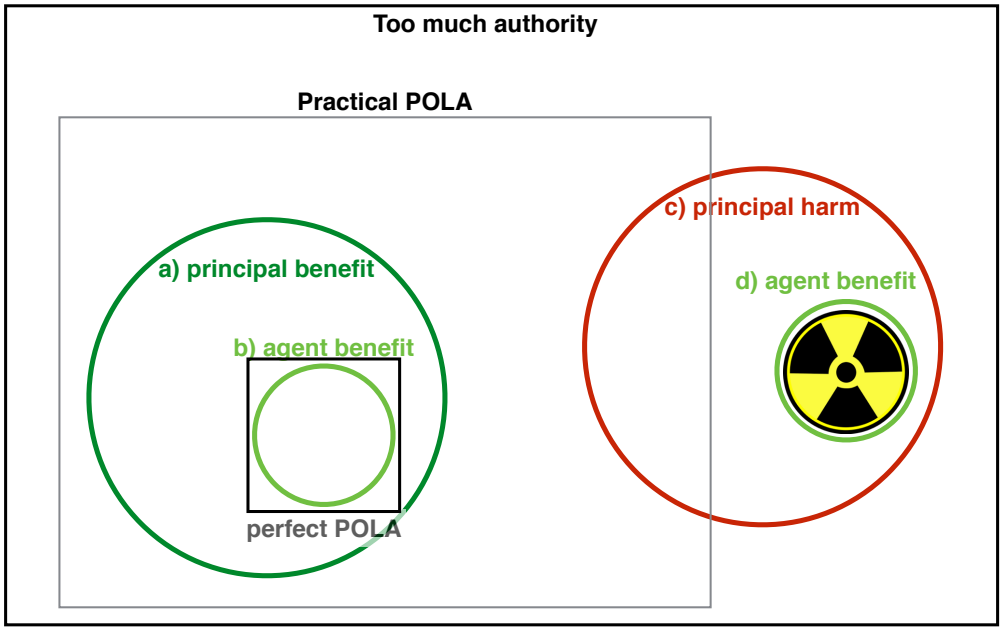


Figure 2 Co-design of incentives and rights.

The outer "too much authority" box represents the rights given to the agent in common practice, such as when programs and imported libraries run with the full privileges of their user. By taking "least" literally, the POLA alternative is often misunderstood as requiring the tiny box labeled "perfect POLA" tightly surrounding circle (b). This requirement is correctly dismissed as too hard to achieve in practice. The perfect is the enemy of the practical.

The box labeled "practical POLA" is large enough to achieve with reasonable effort. It does include agent behaviors that would harm the principal. However, it excludes the region of greatest hazard, where principal harm coincides with agent benefit. When thinking about incentives by themselves, we miss the option to avoid this region by restricting the agent's rights. When thinking about computer security by itself, we miss the differences among agent actions that cause harm to the principal. By thinking about both together, we can co-design principal-agent arrangements in which each technique fills in for weaknesses in the other.

Other surprising opportunities are found examining combinations of the other techniques. By reasoning across both rows and columns, we can help achieve a more cooperative world among diverse human and software, composed together in ever denser networks of principal-agent relationships.

Acknowledgements. We thank Terry A. Stanley and K. Eric Drexler.

References

1 Kathleen M. Eisenhardt. Agency theory: An assessment and review. *The Academy of Management Review*, 14(1):57–74, 1989. URL: `http://www.jstor.org/stable/258191`.

2 Friedrich A. Hayek. The use of knowledge in society. *The American economic review*, pages 519–530, 1945.

3 Friedrich A. Hayek. Competition as a discovery procedure. *New Studies in Philosophy, Politics, Economics and the History of Ideas*, 1978.

4 Laffont Jean-Jacques and David Martimort. The theory of incentives. the principal–agent model. *Princeton, NJ: Princeton Univ*, 2001.

5 Ludwig M. Lachmann. *Capital and its Structure*. Ludwig von Mises Institute, 1956.

6 Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

7 Toby S. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Carmen Lewis, Xin Gao, and Gary Klein. sel4: from general purpose to a proof of information flow enforcement. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 415–429. IEEE, 2013.

8 David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

9 Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

10 Bill Tulloh and Mark S. Miller. Institutions as abstraction boundaries. *Social Learning: Essays in Honor of Don Lavoie*, 2002.

A Calculus with Partially Dynamic Records for Typeful Manipulation of JSON Objects*

Atsushi Ohori[1], Katsuhiro Ueno[2], Tomohiro Sasaki[3], and Daisuke Kikuchi[4]

1 **Research Institute of Electrical Communication, Tohoku University**
Sendai, Miyagi, 980-8577, Japan and
Graduate School of Information Sciences, Tohoku University
Sendai, Miyagi, 980-8579, Japan
ohori@riec.tohoku.ac.jp

2 **Research Institute of Electrical Communication, Tohoku University**
Sendai, Miyagi, 980-8577, Japan and
Graduate School of Information Sciences, Tohoku University
Sendai, Miyagi, 980-8579, Japan
katsu@riec.tohoku.ac.jp

3 **Research Institute of Electrical Communication, Tohoku University**
Sendai, Miyagi, 980-8577, Japan and
Graduate School of Information Sciences, Tohoku University
Sendai, Miyagi, 980-8579, Japan
tsasaki@riec.tohoku.ac.jp

4 **Research Institute of Electrical Communication, Tohoku University**
Sendai, Miyagi, 980-8577, Japan and
Hitachi Solutions East Japan, Ltd.
Sendai, Miyagi, 980-0014, Japan
daisuke.kikuchi.hz@hitachi-solutions.com

Abstract

This paper investigates language constructs for high-level and type-safe manipulation of JSON objects in a typed functional language. A major obstacle in representing JSON in a static type system is their heterogeneous nature: in most practical JSON APIs, a JSON array is a heterogeneous list consisting of, for example, objects having common fields and possibly some optional fields. This paper presents a typed calculus that reconciles static typing constraints and heterogeneous JSON arrays based on the idea of *partially dynamic records* originally proposed and sketched by Buneman and Ohori for complex database object manipulation. Partially dynamic records are dynamically typed records, but some parts of their structures are statically known. This feature enables us to represent JSON objects as typed data structures. The proposed calculus smoothly extends with ML-style pattern matching and record polymorphism. These results yield a typed functional language where the programmer can directly import JSON data as terms having static types, and can manipulate them with the full benefits of static polymorphic type-checking. The proposed calculus has been embodied in SML#, an extension of Standard ML with record polymorphism and other practically useful features. This paper also reports on the details of the implementation and demonstrates its feasibility through examples using actual Web APIs. The SML# version 3.1.0 compiler includes JSON support presented in this paper, and is available from Tohoku University as open-source software under a BSD-style license.

* The first and the second authors have been partially supported by the JSPS KAKENHI Grant Number 25280019 under the title "Foundational research on implementation methods for the ML-style polymorphic programming language SML#". The second author has also been partially supported by the JSPS KAKENHI Grant Number 15K15964 under the title "A systematic approach for developing practical programming languages".

30th European Conference on Object-Oriented Programming (ECOOP 2016).
Editors: Shriram Krishnamurthi and Benjamin S. Lerner; Article No. 18; pp. 18:1–18:25

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1998 ACM Subject Classification D.3.3 [Programming Languages] Language Constructs and Features - data types and structures

Keywords and phrases JSON, Type System, Polymorphic Record Calculus SML#

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.18

1 Introduction

JSON (JavaScript Object Notation) is a text format for serialized structured data. Owing to its simplicity and the human-readable nature, it has become a standard format for data exchanged over the Internet. Web servers and cloud systems, for example, now often provide *JSON API*s, which specify the details of their HTTP requests and responses as JSON formats. Safe and high-level manipulation of JSON objects is, therefore, becoming essential for a programming language to serve as a production language for modern Internet applications.

A common current practice is to provide a library to parse JSON strings and to represent the resulting abstract syntax trees in some data structures of the underlying programming language such as a recursive `datatype` of ML or `JSONObject` class with methods to access JSON attributes with keys. Some languages also provide a mechanism to automatically generate classes and types for given application-specific JSON structures. Examples include JSON schema [17] and the type provider of F# [27]. While these tools free the programmer from the tedious task of deserializing JSON data, from the perspective of a programming language, they are not ideal. Programming with JSON abstract syntax trees is inherently untyped. Type/class generations are meta-level and are not integrated parts of the type system of the language. These approaches cannot take full advantage of high-level and type-safe programming constructs available in advanced programming languages. Indeed, the original motivation for this work came from our experience of developing an ERP system with web interface jointly with a software company [22]. During this development, we had to write codes that dealt with JSON abstract syntax trees. This was not only tedious but also error prone. During this experience we became painfully aware of the need for high-level and type-safe support for JSON objects. The goal of the present paper is to develop programming language constructs for high-level and type-safe manipulation of JSON objects in such a way that they are part of the language type system.

To achieve our goal, we consider a typed functional language with labeled records and pattern matching as an ideal starting point. We note that JSON is a data structure constructed from atomic types (i.e., integers, floating-point numbers, booleans, and strings); "objects," which are named unordered collections of values; and "arrays," which are ordered sequences of values. All of these are common data structures available in a typed functional language. One would therefore expect that they can be directly mapped to typed data structures constructed from labeled records (representing JSON "objects") and lists (representing JSON "arrays"). One would then expect that JSON objects can be directly manipulated through field selection primitives for labeled records and a rich set of higher-order combinators for lists. If such a mapping was indeed possible, then programming with JSON would be a typeful and comfortable programming practice. For example, one would write a function to retrieve, from a JSON object obtained from a weather service on the Internet, a list of cities where the wind speed exceeds 20 mps in the following declarative and concise way:

```
fun highWind weatherMapData =
    foldl (fn ({name, wind, ...}, cities) =>
```

```
        if #speed wind > 20.0 then name::cities else cities)
    nil
    weatherMapData
```

Furthermore, one would expect that such a higher-order function with JSON objects is properly type-checked.

For a data structure constructed with records and lists, this is achieved with record polymorphism [20]. The above function, for example, is given the following static type:

$$\texttt{highWind} \ : \ \forall(t_1 :: \{\texttt{name} : t_2, \texttt{wind} : t_3\}, t_2, t_3 :: \{\texttt{speed} : \texttt{real}\}).\ t_1\ \texttt{list} \to t_2\ \texttt{list}$$

where $\forall(t_1 :: \{\texttt{name} : t_2, \texttt{wind} : t_3\}, \ldots)$ represents kinded abstraction of type variable t_1 whose possible instances are restricted to those record types that contain at least `name` and `wind` fields of type t_2 and t_3. This mechanism ensures type-safe manipulation of lists of records containing `name` and `wind` attributes. Since JSON objects often contain a large collection of complicated record structures, an analogous type-safe and high-level treatment is highly desirable. This mechanism of record polymorphism is an integrated part of SML# [26]. The SML# [26] compiler infers the following typing for the above function:

```
highWind : ['a#{name:'b, wind:'c},'b,'c#{speed:real}. 'a list -> 'b list]
```

This variant of ML provides us an ideal starting point for developing a practical polymorphic language with typeful JSON object manipulation support.

Apparently there are a number of technical issues to be sorted out before developing a satisfactory programming language that realizes this desired view. The major obstacles arise from the property that JSON is inherently heterogeneous. In many practical JSON APIs, objects are required to contain certain set of fields and may contain some optional fields. Moreover, this property is often implicit; objects with optional fields are simply those that contain them, and there is no explicit tag to indicate their existence or nonexistence. Because of this property, JSON arrays are heterogeneous in most cases. This is in sharp contrast with typed representation in ML, where an optional field is represented as `option` datatype, and collection types are always homogeneous. Designing a typed language that uniformly deals with heterogeneous JSON objects constitutes a challenge. In this paper, we provide one solution, and implement it in SML#.

We base our development on the observation made in [6] that a polymorphic record calculus with collection types (set types or list types) can be extended with *partially dynamic records* to form a programming language for high-level and type-safe manipulations of heterogeneous collections. A partially dynamic record type is a refinement of type *dynamic* proposed in [2]. In contrast with the standard dynamic type, however, it statically reveals some record fields. Combining this typing constraint with collection-type constructors, one can obtain a typing mechanism for manipulating heterogeneous persistent collections in an ML-style polymorphic language. In the proposal of [6], the idea and typing rules are sketched out, but their formal properties and implementation methods are not investigated. The goal of the present paper is to develop a typed calculus suitable for JSON data based on the idea of partially dynamic records, to establish type soundness, and to implement the calculus for practical use.

When we separate dynamic typing from partially dynamic records, then types of partially dynamic records intuitively correspond to super types of objects in an object-oriented language. Since a super type also represents a substructure common to all of its subtypes, in a type system with structural subtyping, heterogeneous collections can be represented as (uniform)

collections of a super type of all the possible element types. With a suitable mechanism for dynamic object inspection (dynamic "down casting") for JSON data, an object-oriented type system with object subsumption would certainly be an alternative approach. However, subtyping would complicate polymorphism and type inference with records, and therefore its impact on an implementation method for ML-style functional languages remains to be investigated. Our proposal is suitable for ML-style functional languages and is readily implementable.

Our general motivation of developing static typing for semi-structured external data is shared with the authors of XDuce [14] and CDuce [3]. They aim at developing a typed language for XML processing. In [10], it has been shown that some of these features can be integrated in OCaml. In these systems, however, XML types do not have direct relationship to static types of existing programming languages. As a result, record-like XML structures are not related to labeled records or any other static data structures of the underlying programming language. In the OCaml+XDuce language [10], for example, XML types are opaque types in the OCaml type system. By contrast, our goal is to develop a language mechanism to manipulate external record-like data directly as labeled records in a static type system of ML. A major technical contribution of our research is not just a development of yet another type system for JSON, but seamless and direct integration of JSON structures in a static polymorphic programming language with labeled records.

The specific technical contributions of the present paper include the following:

- We have presented a type system of JSON data and have developed an algorithm to convert JSON data to explicitly typed ones. In the type system, any JSON data have a unique type. The type system contains *partial record types*, and the type reconstruction algorithm computes a partial record type for a heterogeneous JSON array by computing the partial record type that matches all the element types.
- We have developed a calculus with partially dynamic records, defined its operational semantics, and shown a type soundness theorem. By defining the runtime model of partially dynamic records as a pair of a typed JSON object and its *static view*, this calculus serves as a programming language that integrates with JSON data.
- We have presented a compilation method to extend the calculus with a polymorphic primitive to convert a static complex value to dynamic JSON objects. We have also extended the core calculus with ML-style pattern matching for JSON data.
- We have fully implemented the calculus by extending the SML# compiler. Using the implementation, we have evaluated the implemented compiler with a number of examples including a JSON API in a real web service, and have demonstrated its practical feasibility.

The SML# version 3.1.0 includes JSON features presented in this paper, and is available from Tohoku University as open-source software under a BSD-style license. Using this compiler, Tohoku University and Hitachi Solutions East Japan, Ltd. plan to develop a web application framework to be used for development of a personal prescription notebook server for mobile devices.

The rest of the paper is organized as follows. In Section 2, we analyze static properties of JSON data, discuss technical issues in designing a typed functional language with JSON data, and describe our strategy. Section 3 presents the calculus and establishes type soundness. Section 4 describes implementation techniques and reports on our implementation. Section 5 shows some examples using our SML# compiler and demonstrates the feasibility of our proposal. Section 6 compares our proposal with existing work. Section 7 concludes the paper with suggestions for further investigation.

2 Analysis of JSON and our strategy

JSON data, as defined in [5], can be analyzed through the following abstract syntax:

$$j \quad ::= \quad c^b \mid \langle l = j, \ldots, l = j \rangle \mid [j, \ldots, j]$$

c^b is a constant of atomic type b, which represents implicitly typed JSON literals. $\langle l_1 = j_1, \ldots, l_n = j_n \rangle$ is a JSON object, where l ranges over a given set of labels (string literals). In the actual JSON format, an object is an unordered collection of colon-separated name-value pairs. In the presentation of the syntax of the calculus, we use record notation similar to ML records. In $\langle l_1 = j_1, \ldots, l_n = l_n \rangle$, the labels $\{l_1, \ldots, l_n\}$ are pairwise distinct. This property represents the unordered nature of fields in a JSON object. $[j_1, \ldots, j_n]$ is a JSON array consisting of a sequence of JSON values, and corresponds to a list or a vector (immutable array) in a functional language. In this paper, we represent them as lists. Vector views are equally possible.

There is no difficulty in parsing JSON strings and representing the resulting abstract syntax trees as data structures in a programming language. The following is a typical definition in ML:

```
datatype json =
    BOOL of bool | INT of int | REAL of real | STRING of string | NULL
  | OBJECT of (string * json) list
  | ARRAY of json list
```

Some ML compilers provide a library to parse JSON strings into a datatype similar to the above. Our goal is to map untyped runtime values (values of a universal type) as a typed term that can be directly manipulated by a typed ML program.

In the formal development, we omit the null object (`NULL` term above). As we shall briefly explain in Section 4, it can easily be introduced as a built-in constant of a built-in type.

For this data structure we have the following straightforward observation. If we restrict the `ARRAY` variant to be homogeneous, then the above data structure corresponds to runtime values of the following set of ML types:

$$\tau \quad ::= \quad b \mid \{l : \tau, \ldots, l : \tau\} \mid \tau \ \mathit{list}$$

It is a routine matter to define a type reconstruction function that first checks whether a given value of type `json` is typable, and if it is typable, then returns its type. We can then regard type `json` as an instance of type *dynamic* proposed in [2]. This simple observation yields a language having the following features:

- It contains an atomic type `json` (corresponding to *dynamic*) whose runtime values are typed JSON data.
- It provides a language construct of the form `_json` e `as` τ that dynamically checks a typed JSON value whether or not its type component coincides with τ, and if it is the case, then converts the typed JSON value to a value of static type τ. This would raise a runtime exception of *DynamicTypeError* if the type-check fails. An equivalent statement `_typecase` e `of` τ_1 `=>` e_1 `|` $\cdots$ `|` τ_n `=>` e_n `|` `_` `=>` e_0 can also be provided.
- It provides a primitive `import` e to parse a JSON string denoted by e to construct an abstract syntax tree of the JSON data and then to check whether it is typable or not. If it is typable then it creates a typed JSON value. This expression has type `json`.

All the above language constructs can be effectively defined in an ML-style type system, and this approach can straightforwardly yield an extension of ML. While this simple approach

has some usefulness, it is not practical for serious applications with JSON. As we discussed in Introduction, most of practical JSON data are heterogeneous and are rejected by the type-checking process described above.

To analyze the problem, suppose a given JSON array contains objects having mandatory `name:string`, `wind:{speed:real,deg:real}`, `main:{temp:real}` fields, and optional `clouds:{all:int}` and `main:{pressure:int}` fields, such as the following:

```
[
 {"name":"Sendai", "main":{"temp":19.0},
  "wind":{"speed":7.6, "deg":170.0}},
 {"name":"Natori", "main":{"temp":13.0, "pressure":1010},
  "wind":{"speed":5.6, "deg":150.0}, "clouds":{"all":92}},
 ⋮
]
```

This is an example of commonly encountered heterogeneous JSON arrays. A typical program would extract the mandatory `name:string`, `main:{temp:real}`, and `wind:{speed:real}` fields. Since mandatory fields conceptually represent a static constraint on a list of values, we want a programming language for JSON objects to represent the constraint in its static type system so that the programmer can safely use polymorphic function on records with list combinators such as `foldr` and `map`, as suggested in Introduction,

One approach to reconcile the static constraint and heterogeneous JSON arrays is to introduce *partially dynamic records* presented in [6]. A partially dynamic record type, written as $\{\!|l_1 : \tau_1, \ldots, l_n : \tau_n|\!\}$, denotes a dynamically typed record about which it is statically known that its actual type contains the set of fields $l_1 : \tau_1, \ldots, l_n : \tau_n$. A heterogeneous JSON array can then be typed as a list of a partially dynamic record type. For example, the above term can be given a type of the form

$$\{\!|\texttt{name} : \texttt{string},\ \texttt{main} : \{\!|\texttt{temp} : \texttt{real}|\!\},\ \texttt{wind} : \{\!|\texttt{speed} : \texttt{real}, \texttt{deg} : \texttt{real}|\!\}|\!\}\ \texttt{list}$$

indicating the fact that each element of the list is a record that contains at least `name:string`, `main:{temp:real}`, and `wind:{speed:real, deg:real}` fields, and possibly contains more fields. By integrating this typing feature into a static type system with record type, list type, and record polymorphism, we can obtain a typed language that supports high-level and typeful manipulation of JSON objects.

3 Definition of the calculus

This section defines the calculus, establishes its type soundness, and describes the necessary extensions to integrate it into an ML-style language with record polymorphism.

3.1 Typed JSON Objects

We first define a type system for JSON data. We continue to use the abstract syntax of JSON data that we defined in Section 2.

The set of JSON types is given by the following syntax.

$$\pi \quad ::= \quad b \mid \{l : \pi, \ldots, l : \pi\} \mid \{\!|l : \pi, \ldots, l : \pi|\!\} \mid \pi\ \mathit{list} \mid \mathit{json}$$

In addition to JSON object types of the form $\{l_1 : \pi_1, \ldots, l_n : \pi_n\}$ and JSON array types of the form π *list*, we introduce two forms of partial types. One is a partial record type

$$
\begin{array}{rcl}
[\![b]\!] & = & \{c^b \mid c^b \text{ is a constant literal of type } b\} \\
[\![\{l_1 : \pi_1, \ldots, l_n : \pi_n\}]\!] & = & \{\langle l_1 = j_1, \ldots, l_n = j_n\rangle \mid j_1 \in [\![\pi_1]\!], \ldots, j_n \in [\![\pi_n]\!]\} \\
[\![\{\!|l_1 : \pi_1, \ldots, l_n : \pi_n|\!\}]\!] & = & \{\langle l_1 = j_1, \ldots, l_n = j_n, \ldots\rangle \mid j_1 \in [\![\pi_1]\!], \ldots, j_n \in [\![\pi_n]\!]\} \\
[\![\pi\ \mathit{list}]\!] & = & \{[j_1, \ldots, j_n] \mid j_1 \in [\![\pi]\!], \ldots, j_n \in [\![\pi]\!]\} \\
[\![\mathit{json}]\!] & = & \text{the set of all syntactically well formed json terms}
\end{array}
$$

Figure 1 A syntactic model of JSON types.

$\{\!|l_1 : \pi_1, \ldots, l_n : \pi_n|\!\}$, which represents JSON objects that contain at least the set of fields $l_1 : \pi_1, \ldots, l_n : \pi_n$. The other is the type *json* for which no structure is known.

To model the above interpretation of these partial types in a type system, we define the following ordering on the set of types:

- $\pi \leq \mathit{json}$ for any π.
- $\{l_1 : \pi_1, \ldots, l_n : \pi_n, \ldots\} \leq \{\!|l_1 : \pi'_1, \ldots, l_n : \pi'_n|\!\}$ if $\pi_1 \leq \pi'_1, \ldots, \pi_n \leq \pi'_n$.
- $\{\!|l_1 : \pi_1, \ldots, l_n : \pi_n, \ldots|\!\} \leq \{\!|l_1 : \pi'_1, \ldots, l_n : \pi'_n|\!\}$ if $\pi_1 \leq \pi'_1, \ldots, \pi_n \leq \pi'_n$.
- $\pi\ \mathit{list} \leq \pi'\ \mathit{list}$ if $\pi \leq \pi'$.

The reflexive transitive closure of this relation yields a partial ordering on the set of JSON types. This ordering is the converse of the one used in [6], where the ordering is originally introduce to model the amount of information. Here, we use the notation that is familiar in object-oriented type systems [1] for readability. Note, however, that this ordering is not used to define subsumption relation on terms, but to compute the common substructure of element types in a heterogeneous JSON array. The join (least upper bound) of π_1 and π_2 with respect to $\leq$ is written $\pi_1 \sqcup \pi_2$, which denotes the largest common substructure of π_1 and π_2.

The typing rules for JSON data are given below:

$$\vdash c^b : b$$

$$\frac{\vdash j_1 : \pi_1 \quad \cdots \quad \vdash j_n : \pi_n}{\vdash \langle l_1 = j_1, \ldots, l_n = j_n\rangle : \{l_1 : \pi_1, \ldots, l_n : \pi_n\}}$$

$$\frac{\vdash j_1 : \pi_1 \quad \cdots \quad \vdash j_n : \pi_n \quad \pi = \pi_1 \sqcup \cdots \sqcup \pi_n}{\vdash [j_1, \ldots, j_n] : \pi\ \mathit{list}}$$

This type system has a straightforward syntactic model. Figure 1 defines the semantics $[\![\pi]\!]$ of JSON type π. Using this, we define the semantic typing relation, denoted by $\models j : \pi$ if and only if $j \in [\![\pi]\!]$. We can then show the following simple soundness property of the type system:

▶ Proposition 1. If $\vdash j : \pi$ then $\models j : \pi$.

We note that, under this (intended) model, the ordering of JSON types corresponds to set inclusion, i.e., it is the case that if $\pi_1 \leq \pi_2$, then $[\![\pi_1]\!] \subseteq [\![\pi_2]\!]$. Note also that the subsumption rule

$$\frac{\vdash j : \pi \quad \pi \leq \pi'}{\vdash j : \pi'}$$

is sound with respect to the above simple semantics. If we added this rule to our JSON type system, it would correspond to the type system of simple objects with collection types, and our

typing judgment corresponds to minimal typing. As we have noted earlier, subsumption is not used in our system. JSON terms are manipulated through polymorphic record operations, and its polymorphic record typing subsumes the object-oriented subsumption rule. To understand this, suppose we access the `name` field of an object of type `{name:string, age:int}`. Instead of applying subsumption rule to obtain a record type `{name:string}`, we give the `name`-field accessor a polymorphic type so that it can be applicable to any object containing a `name` filed. This approach computes more accurate typing than subtyping.

Therefore we have defined the JSON type system as a relation to deduce a unique (most specific) typing. Moreover, with the type *json*, any JSON term has a unique minimal type in our semantics. Indeed, the following property is easily shown:

▶ Proposition 2. For any j, there is a unique π such that $\vdash j : \pi$. Moreover, if $j \in [\![\pi']\!]$, then $\pi \leq \pi'$.

We define the typed JSON term as a pair $(j : \pi)$ such that $\vdash j : \pi$. We write $\mathit{infer}(j)$ for the typed JSON term $(j : \pi)$ of j.

3.2 The Calculus with Partially Dynamic Records

The idea underlying our calculus for the static manipulation of JSON data is to consider typed JSON objects defined above as runtime values of partially dynamic record types sketched out in [6]. Based on this general idea, this section develops the typed calculus.

The set of terms of the calculus is given below:

$$e \quad ::= \quad c^b \mid x \mid \lambda x.e \mid e\ e \mid \{l = e, \ldots, l = e\} \mid e.l \mid e :: e \mid \mathsf{nil} \mid j \mid (e\ \mathsf{as}\ \pi\ \mathsf{else}\ e)$$

This is a variant of the typed lambda calculus with records and lists, extended with partially dynamic JSON objects and a dynamic type-checking construct. $\{l = e, \ldots, l = e\}$ is a labeled record in Standard ML, and $e.l$ is record field selection that selects the l field from the record e. The operations on lists can be given as primitive functions, and we omit them. j introduces a JSON object as a value of dynamic type. $(e_1\ \mathsf{as}\ \pi\ \mathsf{else}\ e_2)$ dynamically checks whether the type component π' of a typed JSON object e_1 is smaller than or equal to π $(\pi' \leq \pi)$. If it is the case, then it coerces the JSON data to a value of type π; otherwise, it evaluates e_2. When π is a partial record type, then this term denotes a value of a partially dynamic record, or equivalently a partially static JSON object.

This calculus intends to model a polymorphic programming language with JSON manipulation. An actual language requires additional standard features including recursion, recursive datatypes, pattern matching, and exception. There is no difficulty in extending the calculus with these features. In particular, exception is useful in manipulating dynamic objects and we shall implicitly assume that exception mechanism is available in writing examples.

The set of types is given below:

$$\tau \quad ::= \quad b \mid \tau \to \tau \mid \{l : \tau, \ldots, l : \tau\} \mid \{\!|l : \tau, \ldots, l : \tau|\!\} \mid \tau\ \mathit{list} \mid \mathit{json}$$

Note that this set of types subsumes all the JSON types (ranged over by π). $\{l : \tau, \ldots, l : \tau\}$ is a labeled record type, and τ *list* is a list type; when they appear in a typing judgment $\Gamma \vdash e : \tau$, they denote ML records and ML lists. They also denote JSON objects and JSON arrays when they appear as type specifications π in a term $(e\ \mathsf{as}\ \pi\ \mathsf{else}\ e)$.

Let Γ range over the set of type assignment, which is a finite function from variables to types. $\mathit{dom}(\Gamma)$ is the domain of Γ. For a typing assignment Γ, $\Gamma\{x : \tau\}$ is Γ' such that $\mathit{dom}(\Gamma') = \mathit{dom}(\Gamma) \cup \{x\}$, $\Gamma'(x) = \tau$, and $\Gamma'(y) = \Gamma(y)$ for any $y \in \mathit{dom}(\Gamma)$ such that $x \neq y$.

$$\Gamma \vdash c^b : b \qquad \Gamma \vdash \mathsf{nil} : \tau\ \mathit{list} \quad (\text{for any } \tau) \qquad \Gamma \vdash j : \mathit{json}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma\{x : \tau\} \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau\ \mathit{list}}{\Gamma \vdash e_1 :: e_2 : \tau\ \mathit{list}} \qquad \frac{\Gamma \vdash e_i : \tau_i \quad (1 \le i \le n)}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}$$

$$\frac{\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_i : \tau_i, \ldots, l_n : \tau_n\} \quad (1 \le i \le n)}{\Gamma \vdash e.l_i : \tau_i}$$

$$\frac{\Gamma \vdash e : \{\!|l_1 : \tau_1, \ldots, l_i : \tau_i, \ldots, l_n : \tau_n|\!\} \quad (1 \le i \le n)}{\Gamma \vdash e.l_i : \tau_i}$$

$$\frac{\Gamma \vdash e_1 : \mathit{json} \quad \Gamma \vdash e_2 : \pi}{\Gamma \vdash (e_1\ \mathsf{as}\ \pi\ \mathsf{else}\ e_2) : \pi} \qquad \frac{\Gamma \vdash e_1 : \{\!|l_1 : \tau_1, \ldots, l_n : \tau_n|\!\} \quad \Gamma \vdash e_2 : \pi}{\Gamma \vdash (e_1\ \mathsf{as}\ \pi\ \mathsf{else}\ e_2) : \pi}$$

Figure 2 Type system for the calculus with partially dynamic records.

$$\begin{aligned}
\mathcal{R}(c^b, b) &= c^b \\
\mathcal{R}(j, \mathit{json}) &= \mathit{infer}(j) \\
\mathcal{R}([j_1, \ldots, j_n], \pi\ \mathit{list}) &= \mathcal{R}(j_1, \pi) :: \cdots :: \mathcal{R}(j_n, \pi) :: \mathsf{nil} \\
\mathcal{R}(\langle l_1 = j_1, \ldots, l_n = j_n \rangle, \{l_1 : \pi_1, \ldots, l_n : \pi_n\}) &= \{l_1 = \mathcal{R}(j_1, \pi_1), \ldots, l_n = \mathcal{R}(j_n, \pi_n)\} \\
\mathcal{R}(\langle l_1 = j_1, \ldots, l_i = j_i, \ldots, l_n = j_n \rangle, \{\!|l_1 : \pi_1, \ldots, l_i : \pi_i|\!\}) &= \\
(\mathit{infer}(\langle l_1 = j_1, \ldots, l_i = j_i, \ldots, l_n = j_n \rangle), \{l_1 &= \mathcal{R}(j_1, \pi_1), \ldots, l_i = \mathcal{R}(j_i, \pi_i)\})
\end{aligned}$$

Figure 3 Dynamic coercion of JSON data.

The type system is defined as a set of rules to derive the typing relation of the form $\Gamma \vdash e : \tau$ indicating that expression e has type τ under type assignment Γ. The set of typing rules of the calculus is given in Figure 2.

The observant reader may have noticed that in the pure calculus defined above, there is no closed term having a partially dynamic record type, since $(e_1\ \mathsf{as}\ \pi\ \mathsf{else}\ e_2)$ requires else term e_2. As we shall explain later in Section 4, a typical use of as construct is of the form $(j\ \mathsf{as}\ \pi\ \mathsf{else}\ \mathit{DynamicTypeError})$, where *DynamicTypeError* is a term to raise an exception when j's runtime type does not match with type π. This term has type π for any π.

We construct a big-step operational semantics in the style of natural semantics [18]. The set of runtime values, ranged over by v, is given below:

$$v \quad ::= \quad c^b \mid \mathit{Cls}(E, x, e) \mid \{l = v, \ldots, l = v\} \mid v :: v \mid \mathsf{nil} \mid (j : \pi) \mid (j : \pi, v) \mid \mathit{Wrong}$$

$\mathit{Cls}(E, x, e)$ is a function closure. $(j : \pi)$ is a typed JSON value defined in Subsection 3.1. $(j : \pi, v)$ is a pair of a typed JSON value and its *view* v. *Wrong* indicates runtime failure.

A typed JSON value $(j : \pi)$ is a runtime model of type *json*. A typed JSON value with static view $(j : \pi, v)$ is a runtime model of partially dynamic JSON terms, i.e., those terms whose types contain partial record types. These (partially) dynamic values are type-checked at runtime by the construct $(e_1\ \mathsf{as}\ \pi\ \mathsf{else}\ e_2)$, as explained above. Figure 3 gives an algorithm $\mathcal{R}$, which takes a typed JSON value and a type and returns a value of that type.

We write $(j : \pi) \approx v$ if v is identical to $(j : \pi)$ or is of the form $(j : \pi, v')$ for some v'. Let E range over the set of value assignments, which is a finite function from variables to values. Figure 4 shows the set of evaluation rules that derives the evaluation relation of the

$$E \vdash c^b \Downarrow c^b \qquad E \vdash \mathsf{nil} \Downarrow \mathsf{nil} \qquad E \vdash j \Downarrow \mathit{infer}(j) \qquad E \vdash \lambda x.e \Downarrow \mathit{Cls}(E, x, e)$$

$$\frac{E(x) = v}{E \vdash x \Downarrow v} \qquad \frac{E \vdash e_1 \Downarrow \mathit{Cls}(E', x, e) \quad E \vdash e_2 \Downarrow v' \quad E'\{x : v'\} \vdash e \Downarrow v}{E \vdash e_1\ e_2 \Downarrow v}$$

$$\frac{E \vdash e_1 \Downarrow v_1 \quad E \vdash e_2 \Downarrow v_2}{E \vdash e_1 :: e_2 \Downarrow v_1 :: v_2} \qquad \frac{E \vdash e_1 \Downarrow v_1 \quad \cdots \quad E \vdash e_n \Downarrow v_n}{E \vdash \{l_1 = e_1, \ldots, l_n = e_n\} \Downarrow \{l_1 = v_1, \ldots, l_n = v_n\}}$$

$$\frac{E \vdash e \Downarrow \{l_1 = v_1, \ldots, l_i = v_i, \ldots, l_n = v_n\} \quad (1 \le i \le n)}{E \vdash e.l_i \Downarrow v_i}$$

$$\frac{E \vdash e \Downarrow (j : \pi, \{l_1 = v_1, \ldots, l_i = v_i, \ldots, l_n = v_n\}) \quad (1 \le i \le n)}{E \vdash e.l_i \Downarrow v_i}$$

$$\frac{E \vdash e_1 \Downarrow v \quad (j : \pi') \approx v \quad \pi' \le \pi}{E \vdash (e_1\ \mathsf{as}\ \pi\ \mathsf{else}\ e_2) \Downarrow \mathcal{R}(j, \pi)} \qquad \frac{E \vdash e_1 \Downarrow v \quad (j : \pi') \approx v \quad \pi' \not\le \pi \quad E \vdash e_2 \Downarrow v'}{E \vdash (e_1\ \mathsf{as}\ \pi\ \mathsf{else}\ e_2) \Downarrow v'}$$

Figure 4 Operational semantics of the calculus.

form $E \vdash e \Downarrow v$ indicating that e evaluates to v under value assignment E. We note that the entire set of rules should be taken with the following implicit rules yielding *Wrong*: if any of the conditions in the premises are not satisfied or evaluation of any subterm yields *Wrong*, then the entire evaluation yields *Wrong*.

3.3 Soundness

We first define value typing, denoted by $\models v : \tau$, indicating that runtime value v has type τ as the following relation:

- $\models c^b : b$
- $\models \mathit{Cls}(E, x, e) : \tau_1 \to \tau_2$ iff there is some Γ such that $\models E : \Gamma$ and $\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2$.
- $\models \{l_1 = v_1, \ldots, l_n = v_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ iff $\models v_i : \tau_i$ for each $1 \le i \le n$.
- $\models \mathsf{nil} : \tau\ \mathit{list}$ for any τ.
- $\models v_1 :: v_2 : \tau\ \mathit{list}$ iff $\models v_1 : \tau$ and $\models v_2 : \tau\ \mathit{list}$.
- $\models (j : \pi) : \mathit{json}$ iff $\vdash j : \pi$ in the JSON type system.
- $\models (j : \pi, v) : \{\!|l_1 : \pi_1, \ldots, l_n : \pi_n|\!\}$ iff $\vdash j : \pi$ in the JSON type system, $\pi \le \{\!|l_1 : \pi_1, \ldots, l_n : \pi_n|\!\}$, and $\models v : \{l_1 : \pi_1, \ldots, l_n : \pi_n\}$.

We write $\models E : \Gamma$ if $\mathit{dom}(E) = \mathit{dom}(\Gamma)$ and $\models E(x) : \Gamma(x)$ for any $x \in \mathit{dom}(E)$.

The following property is easily shown from the semantics of JSON typing:

▶ **Lemma 1.** *If* $\vdash j : \pi$ *and* $\pi \le \pi'$, *then* $\models \mathcal{R}(j, \pi') : \pi'$.

We then have the following:

▶ **Theorem 2.** *If* $\Gamma \vdash e : \tau$, $\models E : \Gamma$ *and* $E \vdash e \Downarrow v$, *then* $\models v : \tau$.

Proof. This is proved by induction on the size of derivations of $E \vdash e \Downarrow v$. Here we only show a few cases involving JSON values.

Case $e = j$. We have $\models \mathit{infer}(j) : \mathit{json}$ by the definition of $\mathit{infer}(j)$.

Case $e = (e_1\ \mathsf{as}\ \pi\ \mathsf{else}\ e_2)$. From the evaluation rules, we have $E \vdash e_1 \Downarrow v_1$. From the typing rules, we have either $\Gamma \vdash e_1 : \mathit{json}$ or $\Gamma \vdash e_1 : \{\!|\cdots|\!\}$. By the induction hypothesis, we have either $\models v_1 : \mathit{json}$ or $\models v_1 : \{\!|\cdots|\!\}$. From the definition of value typing, v_1 is either

$(j : \pi')$ or $(j : \pi', v')$ for some j, π', and v' such that $\vdash j : \pi'$. In both cases, $(j : \pi') \approx v_1$ holds. If $\pi' \leq \pi$, we have $\models \mathcal{R}(j, \pi) : \pi$ by Lemma 1. If $\pi' \not\leq \pi$, we have $\Gamma \vdash e_2 : \pi$ and $E \vdash e_2 \Downarrow v'$ from the typing and evaluation rules. By the induction hypothesis, we have $\models v' : \pi$. □

The calculus so far defined is a minimal one with a monomorphic type system. In the rest of this section, we describe some extensions to develop a polymorphic functional language.

3.4 Extension to polymorphism

There is no difficulty in extending the monomorphic type system of the calculus defined above to ML-style polymorphism. Because the runtime models of both dynamic type *json* and partial record types $\{\!|l_1 : \pi_1, \ldots, l_n : \pi_n|\!\}$ are JSON data, it is natural to restrict the target type π of $(e_1 \text{ as } \pi \text{ else } e_2)$ to be a monomorphic JSON type. With this restriction, the type system straightforwardly extends to an ML-style polymorphic type system.

Moreover, since partially dynamic JSON data are viewed as ML records, the calculus nicely blends with record polymorphism. Indeed, we can extend a polymorphic record calculus [20] to JSON data and partial record field selections without much difficulty. In our implementation we shall report in Section 4, a partial record field selection is written as a function

```
fn x => #foo (view x)
```

which has a record-polymorphic type

```
['a#{foo: 'b}, 'b. 'a dyn -> 'b]
```

indicating the fact that it accepts any partially dynamic record that has at least a `foo` field.

3.5 Serializing ML values as JSON data

The calculus defined above captures how to deal with given JSON data in a statically typed language. In our calculus, we have included JSON term j as a new syntactic category. Our implicit assumption underlying this design is that JSON term j is an abstract syntax of the JSON format. From a practical perspective, this corresponds to including a primitive to parse a string representation of JSON data written by the programmer or received through an I/O primitive. This is what we have done in our implementation.

The opposite direction is of course necessary; we also want to serialize values in the language to generate JSON data in a type-consistent way. This problem can be stated as a problem to define a construct `toJson`(e) that produces a JSON value j such that (`toJson`(e) as π else $\bot$) is equivalent to e. Moreover, we would like `toJson`() to work polymorphically. This is a sub-problem of type-dependent value printing.

One approach is to add primitives to introduce term of *dynamic* type in [2] and to format a dynamic value. However, there is a subtle technical issue concerning the introduction of *dynamic* type. In practical implicitly typed polymorphic languages in the ML family, only the compiler knows the type information; therefore, π does not exist at runtime. Recursive data types make things more difficult since the recursive structure of a data type is not necessarily determined from its name; for example, *list* is just a type name and hence does not indicate its recursive structure consisting of cons and nil. We have partially solved this problem for monomorphic ground types and have implemented a special construct that embeds type representation in object codes. This enables us to *reify* a type of expression. Using this, SML# implements the following primitive:

```
dynamic : ['a. 'a -> dynamic]
```

The current limitation is that this primitive can reify the type information only when the type of the argument to this primitive is monomorphic. Details of this technique is outside the scope of this paper, and we will present them elsewhere.

On top of this dynamic primitive, we introduce the following two primitives:

```
toDynamic : ['a#json. 'a -> dynamic]
dynamicToJson : dynamic -> {json:void dyn, string:string}
```

`toDynamic` is an alias to `dynamic` whose argument type is constrained to JSON type π through the kind constraint `'a#json` that restricts instance of `'a` to JSON types. `dynamicToJson` converts dynamic value to dynamic json object and its string representation. Using them, we can write a code

```
let
  val x = [{name = "Joe", score = {math = 76, lang = 89}},
           {name = "Bob", score = {math = 96, lang = 94}}]
in
  print (#string (dynamicToJson (toDynamic x)))
end
```

that produces the following output in JSON format:

```
[
 {"name":"Joe", "score":{"lang":89, "math":76}},
 {"name":"Bob", "score":{"lang":94, "math":96}}
]
```

3.6 Pattern matching with JSON

Another useful extension is to integrate JSON manipulation constructs (e_1 `as` π `else` e_2) with ML-style pattern matching. This integration relieves the programmer from writing type annotations, and enables the programmer to use familiar programming idioms with ML-style patterns. This extension can be provided by syntactic elaboration in a language with ML-style pattern matching. Here we outline the elaboration.

We assume that the core calculus is extended with the ML-style case statement

`case` exp `of` pat_1 `=>` exp_1 `|` $\cdots$ `|` pat_n `=>` exp_n

with an extension that record patterns match with partially dynamic JSON objects as well as ML records. We then introduce the following case statement for JSON data

`jsonCase` e `of` jp_1 `=>` exp_1 `|` $\cdots$ `|` jp_n `=>` exp_n

that performs a case analysis on dynamic or partially dynamic JSON value e against the set of JSON patterns $jp_1, \ldots, jp_n$. The set of JSON patterns is given below:

jp	::=	c^b	(constants of type b)
	|	$x : \pi$	(any JSON value of π)
	|	$\{l = jp, \ldots, l = jp\}$	(JSON object)
	|	$\{l = jp, \ldots, l = jp, \ldots\}$	(partial JSON object)
	|	$\mathsf{nil} : \pi\ list$	(empty JSON array)
	|	$jp :: x$	(JSON array)

$$
\begin{array}{rcl}
Pat(c^b) & = & c^b \\
Pat(x : \pi) & = & x \\
Pat(\{l_1 = jp_1, \ldots, l_n = jp_n\}) & = & \{l_1 = Pat(jp_1), \ldots, l_n = Pat(jp_n)\} \\
Pat(\{l_1 = jp_1, \ldots, l_n = jp_n, \ldots\}) & = & \{\!|l_1 = Pat(jp_1), \ldots, l_n = Pat(jp_n)|\!\} \\
Pat(\mathsf{nil} : \pi\ list) & = & \mathsf{nil} \\
Pat(jp_1 :: x) & = & Pat(jp_1) :: x
\end{array}
$$

$$
\begin{array}{rcl}
Ty(c^b) & = & b \\
Ty(x : \pi) & = & \pi \\
Ty(\{l_1 = jp_1, \ldots, l_n = jp_n\}) & = & \{l_1 : Ty(jp_1), \ldots, l_n : Ty(jp_n)\} \\
Ty(\{l_1 = jp_1, \ldots, l_n = jp_n, \ldots\}) & = & \{\!|l_1 : Ty(jp_1), \ldots, l_n : Ty(jp_n)|\!\} \\
Ty(\mathsf{nil} : \pi\ list) & = & \pi\ list \\
Ty(jp_1 :: x) & = & Ty(jp_1)\ list
\end{array}
$$

Figure 5 The type-part and the pattern-part of JSON pattern.

This definition is similar to that of Standard ML pattern language, except that variables and nil must be type-annotated to ensure that the pattern corresponds to a unique JSON type.

The above jsonCase expression is translated to a combination of JSON primitives of the core calculus and case expressions of ML. To define the translation scheme, we define the type part $Ty(jp)$ of jp and the pattern part $Pat(jp)$ of jp in Figure 5. JSON case expression of the form

$$\mathsf{jsonCase}\ e\ \mathtt{of}\ jp_1\ \mathtt{=>}\ e_1\ \mathtt{|}\ \cdots\ \mathtt{|}\ jp_n\ \mathtt{=>}\ e_n$$

is translated to the following nested case analysis term E_1:

$$
\begin{array}{rcl}
E_1 & = & \mathsf{case}\ (e\ \mathsf{as}\ Ty(jp_1)\ \mathsf{else}\ E_2)\ \mathtt{of}\ Pat(jp_1)\ \mathtt{=>}\ e_1\ \mathtt{|}\ \mathtt{\_}\ \mathtt{=>}\ E_2 \\
E_2 & = & \mathsf{case}\ (e\ \mathsf{as}\ Ty(jp_2)\ \mathsf{else}\ E_3)\ \mathtt{of}\ Pat(jp_2)\ \mathtt{=>}\ e_2\ \mathtt{|}\ \mathtt{\_}\ \mathtt{=>}\ E_3 \\
 & \vdots & \\
E_n & = & \mathsf{case}\ (e\ \mathsf{as}\ Ty(jp_n)\ \mathsf{else}\ DynamicTypeError) \\
 & & \mathtt{of}\ Pat(jp_n)\ \mathtt{=>}\ e_n\ \mathtt{|}\ \mathtt{\_}\ \mathtt{=>}\ MatchError
\end{array}
$$

where *MatchError* and *DynamicTypeError* are terms to raise exceptions. *DynamicTypeError* indicates that dynamic type-checking fails. *MatchError* indicates that list or constant pattern matching in jp_i fails.

4 Implementation

We have implemented all the features presented in the previous section in the SML# compiler version 3.1.0, which is available from: `http://www.pllab.riec.tohoku.ac.jp/smlsharp/`. The JSON features are supported both in the separate compilation mode and in the interactive session. The following is a very simple actual interactive session.

```
$ smlsharp
SML# 3.1.0 ··· for x86_64-pc-linux-gnu with LLVM 3.7.1
# open JSON;
  ··· output message on JSON structure being opened.
# import "{\"name\":\"SML#\", \"version\":\"3.1\"}";
val it = _ : void dyn
# _jsoncase it of {name=x:string, ...} => x;
val it = "SML#" : string
```

SML# interpreter prompts the user by printing "`#`". `open JSON` makes the primitives defined in the `JSON` structure, such as `import`, available at the top-level. The user can also write `JSON.import` without opening `JSON` structure. Section 5 shows more examples.

Through our effort of extending the full-fledged and complex compiler with JSON support, we have observed that we can implement the required dynamic typing systematically and efficiently if we provide a mechanism for the compiler to access user-level library codes. Based on this observation, we have successfully completed our implementation using user-level SML# codes with relatively small amount of modification to the compiler.

The implementation consists of three components. The first is the library for JSON object manipulation written as user-level codes. The second is typed elaboration that compiles $(e \text{ as } \pi \text{ else } e)$ using type information. The third is syntactic elaboration that transforms jsonCase expression. The compiler performs the second and the third transformation using the JSON support library through the mechanism to access user-level codes. The compiler support is necessary for two reasons: (1) in both translations, the generated codes are type-varying and therefore the translation codes are not typable, and (2) the typed elaboration requires compile-time static type information.

We report on its details in the following three steps. First, we explain the strategy of typed elaboration that implements the *json* type with its introduction and elimination. Second, we extend the typed elaboration with the partially dynamic records. Last, we describe the techniques we used in implementing the jsonCase translation presented in Subsection 3.6.

4.1 Implementation of *json* type

Implementation of the *json* type involves the introduction of the JSON term, denoted by j in the calculus, and the implementation of the $(e_1 \text{ as } \pi \text{ else } e_2)$ statement that eliminates type *json*. We first review the semantics structure defined in the formal calculus with our strategy to implement them, and then describe the details of our implementation.

A semantic object $(j : \pi)$ of type *json* is a pair of a JSON term and a JSON type. The structure of j and π are represented as ML datatypes. The introduction of a JSON term j of type *json* is implemented as an ML function that parses a given JSON string to obtain a JSON term j, infers its JSON type π such that $\vdash j : \pi$, and then returns the pair of the parse result and π.

Those JSON terms are dynamically type-checked and converted to ML value through the construct $(e_1 \text{ as } \pi \text{ else } e_2)$. We provide the syntax

`_json` e `as` π

for $(e_1 \text{ as } \pi \text{ else } \mathit{DynamicTypeError})$ where else term e_2 is fixed to the one that raise exception. To implement this, we need to *reify* π. For this purpose, a datatype representation of JSON type π is defined in the user-level library and the compiler generates user-level code representing π in that representation. This code generation are done by searching for

appropriate user-level codes in the library from the given context and inserting references to them. Then, the compiler translates `_json` e `as` π into the user-level code that extracts the JSON-type part π' from the result of evaluation of e, checks whether $\pi' \leq \pi$ holds, and if it succeeds then converts the JSON term of e to a runtime value of type π. The resulting code constitutes a composition of user-level functions in the library, each of which performs one of the above steps. The same searching mechanism as the reification is used to generate this composition.

In actual implementation, JSON types are represented by the following definition:

```
datatype jsonTy =
    BOOLty | INTty | REALty | STRINGty | NULLty
  | ARRAYty of jsonTy
  | RECORDty of (string * jsonTy) list
  | PARTIALRECORDty of (string * jsonTy) list
  | JSONty
```

To represent unordered record fields, the list of `RECORDty` and `PARTIALRECORDty` are sorted in alphabetical order. `NULLty` is the type of the null value of JSON, whose treatments we omitted in the formal development in Section 3.

We implement the *json* type as a type-annotated JSON term rather than the pair of a JSON term and JSON type as mentioned above, so that the JSON type of any subterms of JSON are computed once JSON is read. The following shows the definition of the type-annotated JSON and type *json* whose name in implementation is `dyn`:

```
datatype json =
    BOOL of bool | INT of int | REAL of real | STRING of string | NULL
  | ARRAY of json list * jsonTy
  | OBJECT of (string * json) list
datatype dyn = DYN of json
```

In addition to the standard JSON term representation, it is sufficient to add a type annotation to the JSON array; the other component carries type information. The data constructor `DYN` is hidden from the user; hence the `dyn` type can be seen as atomic types from the user.

We provide a function of the following signature for the user to read JSON data as a value of type `dyn`:

```
val import : string -> dyn
```

This function parses JSON in the given string and computes the element type of every array appearing in the given JSON.

We define the following user-level functions for the compiler to use during compilation:

```
val getJson : dyn -> json
val checkTy : json * jsonTy -> unit
val checkInt : json -> int
val checkString : json -> int
 ...
val checkArray : json -> json list
```

`getJson` e returns the internal JSON term of internal type `json`. `checkTy(e,` π`)` extracts the type π' of JSON term e and if $\pi' \not\leq \pi$ then it raises `RuntimeTypeError` exception. `checkInt`, `checkString`, `checkArray`, etc., are coercion functions defined for atomic types

`int`, `string`, etc., and for type constructors `array` and others. Each coercion function checks the JSON type of a given JSON term and converts it to the corresponding ML value.

Using these user-level functions, we define the following two code generation functions in the compiler: `coerceJson(`e`,`π`)` to call appropriate `check` function to obtain an ML value from a JSON term e, and `tyToJsonTy(`π`)` to convert a static type π to a term of type `jsonTy`. These definitions are straightforward except in the case of partially dynamic records we will mention in the next subsection. `_json` e `as` π is compiled by the following code generation function:

```
fun compileJson (e, ty) =
    let
      val jsonExp = App (J.getJson(), [e])
      val viewExp = coerceJson (jsonExp, ty)
      val viewTy = tyToJasonTy ty
      val checkExp = App (J.checkTy(), [jsonExp, viewTy])
    in
      Seq [checkExp, Typed (viewExp, ty)]
    end
```

In this code, `App`, `Seq`, and `Typed` generates application term, sequencing term, and type annotated term, respectively. `J.getJson()` and `J.checkTy()` are references to the user-level functions of the same name through the searching mechanism mentioned above.

4.2 Implementation of partially dynamic records

The approach presented in the previous subsection is extended to partially dynamic records. In the calculus presented in Section 3, a partially dynamic record is regarded as a value of type dynamic coupled with its view. To deal with partially dynamic records and dynamic values uniformly, we extend `dyn` type to `'a dyn` where `'a` is the type of the view. We additionally introduce a new type `void` indicating that no view is available; an attempt to view a term of type `void dyn` will result in `AttemptToReturnVOIDValue` runtime exception. With these extensions, *json* is represented as `void dyn`, and $\{\!|l_1 : \tau_1, \ldots, l_n : \tau_n|\!\}$ is represented as {l_1`:`τ_1`,` ..., l_n`:`τ_n`}` `dyn`. For example, to coerce a JSON object e to a list of partially dynamic records of type $\{\!|$`name : string`$|\!\}$, the programmer can write the following:

```
_json e as {name: string} dyn list
```

then an ML list of partially dynamic records is obtained by generating the view of each element of the list.

One concern of this approach is performance; the calculus produces a view for a partially dynamic record every time a partially dynamic record is generated. Naive implementation would waste time and memory by producing unused views when reading a large JSON array consisting of a variety of JSON objects. To avoid this overhead, we delay the generation of views until the view is really needed. The definition of `dyn` is eventually as follows:

```
datatype 'a dyn = DYN of (json -> 'a) * json
```

This `'a` is to be instantiated to either a record type or `void`. The function of type `json -> 'a` produces an ML record of type `'a` as a view of the given JSON term. Partially dynamic records are then converted to ML records by the following function:

```
val view : 'a dyn -> 'a = fn DYN (f, x) => f x
```

The case of `coerceJson` for partially dynamic record types is given below:

```
fun coerceJson (jsonExp, DYNty argTy) =
    let
      val funExp = Fn (fn x => coerceJson (Var x, argTy))
      val jsonTy = tyToJsonTy argTy
    in
      App (J.makeCoerce(), [jsonExp, jsonTy, funExp])
    end
```

`Fn` is a compiler primitive to convert a meta-level function to the corresponding object-level function term. `makeCoerce : json -> jsonTy -> (json -> 'a) -> 'a dyn` is a user-level function that takes json term j, its type, and a coercion function, and generates a partially dynamic term.

4.3 Implementation of pattern matching with JSON

We have implemented the following syntax that embodies jsonCase presented in Subsection 3.6:

$$\texttt{\_jsoncase}\ e\ \texttt{of}\ jp_1\ \texttt{=>}\ e_1\ \texttt{|}\ \cdots\ \texttt{|}\ jp_n\ \texttt{=>}\ e_n$$

with several shorthands available in ML's pattern language. Moreover, along with the syntax of JSON, we allow any string literals to occur in jp as labels of record fields. An example of string literals as record labels appears in Subsection 5.2.

The elaboration scheme presented in Subsection 3.6 cannot straightforwardly incorporate our implementation strategy owing to the following issues: (1) the ML record pattern does not match with partially dynamic records since partially dynamic records have their own data constructors, (2) the construction of the view of a partially dynamic record is delayed by a function, and (3) the code duplication in the present elaboration scheme increases the code size exponentially. A standard solution to these issues would be to develop a match compilation algorithm for jsonCase, similarly to ML's pattern matching compilation.

Instead of taking the full-fledged approach of developing a match compiler, we adopt the following light-weight strategy. We translate `_jsoncase` into a nested `case` expression that interleaves pattern matching with view construction of partially dynamic records. Let m be the number of (top-level) partially dynamic record patterns of the form $\{l_j^1{=}p_j^1, \ldots, l_j^k{=}p_j^k, \ldots\}$ $(1 \leq j \leq m)$ occurring in jp_i. The result $\mathcal{T}(e, jp_i\ \texttt{=>}\ e_i)$ of translation of match $jp_i\ \texttt{=>}\ e_i$ is the nested `case` expression of the form

$$\texttt{case}\ e\ \texttt{of}\ \overline{jp_i}\ \texttt{=>}\ E_1\ \texttt{|}\ \texttt{\_}\ \texttt{=>}\ \texttt{raise}\ M$$

where $\overline{jp_i}$ is the pattern obtained by replacing the j-th partially dynamic record pattern with a fresh variable x_j, M is the exception indicating the fact that the other cases should be tried. The main expression E_1 is generated by the following cascading equations:

$$\begin{aligned}
E_1 \quad = \quad & \texttt{case view}\ x_1\ \texttt{of}\ \{l_1^1{=}x_1^1, \ldots, l_1^k{=}x_1^k\}\ \texttt{=>} \\
& \quad \mathcal{T}(x_1^1, jp_1^1\ \texttt{=>}\ \mathcal{T}(\cdots\ \mathcal{T}(x_1^k, jp_1^k\ \texttt{=>}\ E_2)\cdots)\cdots) \\
& \texttt{|}\ \texttt{_}\ \texttt{=>}\ \texttt{raise}\ M \\
\vdots & \\
E_m \quad = \quad & \texttt{case view}\ x_m\ \texttt{of}\ \{l_m^1{=}x_m^1, \ldots, l_m^l{=}x_m^l\}\ \texttt{=>} \\
& \quad \mathcal{T}(x_m^1, jp_m^1\ \texttt{=>}\ \mathcal{T}(\cdots\ \mathcal{T}(x_m^1, jp_m^l\ \texttt{=>}\ e_i)\cdots)\cdots) \\
& \texttt{|}\ \texttt{_}\ \texttt{=>}\ \texttt{raise}\ M
\end{aligned}$$

The fallback to the next match is realized by handling the exception M. The entire `_jsoncase` is translated into the following expression:

```
let
  val x = e
  exception M
in
  T(_json x as Ty(jp1), jp1 => e1) handle M =>
      ⋮
  T(_json x as Ty(jpn), jpn => en) handle M =>
  raise Match
end
```

In addition to patterns presented in Subsection 3.6, we have implemented a pattern for heterogeneous lists of the following form:

$$jp \quad ::= \quad \cdots \mid jp_1 \texttt{ :: } \cdots \texttt{ :: } jp_n \texttt{ :: } x$$

This pattern matches with the first n elements of the given JSON array and binds x to the rest of the array. In contrast to ML's list pattern, the type of each element jp_i of a heterogeneous list pattern may differ. The matching with this heterogeneous list pattern is performed by coercing the given JSON term to the term of type `void dyn list`, taking n elements from the head of the list, and then matching the i-th element with jp_i. To realize this, our implementation translates the following case:

$$\texttt{\_jsoncase } e_1 \texttt{ of } jp_1 \texttt{ :: } \cdots \texttt{ :: } jp_n \texttt{ :: } x \texttt{ => } e_2$$

into the following expression:

```
case (_json e1 as void dyn list) of
  x1::⋯::xn::x =>
    _jsoncase x1 of jp1 =>
        ⋮
      _jsoncase xn of jpn => e2
```

where $x_1, \ldots, x_n$ are fresh variables.

5 Evaluations through realistic examples

In this section, we demonstrate the feasibility and the usefulness of our approach through realistic examples using existing Web APIs.

5.1 Heterogeneous record collections in JSON

We first go through the basic usage of our JSON extension using simple examples.

A typical usage of JSON is to represent a collection of objects. The following example imports a JSON array of objects, coerces it to a list of records, and extracts their `name` fields:

```
val J = "[{\"name\":\"Joe\", \"age\":21, \"grade\":1.1},
          {\"name\":\"Sue\", \"age\":31, \"grade\":2.0},
          {\"name\":\"Bob\", \"age\":41, \"grade\":3.9}]"
```

```
# val J = "[{\"name\":\"Joe\", \"age\":21, \"grade\":1.1},\
          \ {\"name\":\"Sue\", \"age\":31, \"grade\":2.0},\
          \ {\"name\":\"Bob\", \"age\":41, \"grade\":3.9}]";
val J =
  "[{\"name\":\"Joe\", \"age\":21, \"grade\":1.1},
    {\"name\":\"Sue\", \"age\":31, \"grade\":2.0},
    {\"name\":\"Bob\", \"age\":41, \"grade\":3.9}]" : string
# fun getNames l = map #name l;
val getNames = fn : ['a#{name: 'b}, 'b. 'a list -> 'b list]
# val j = import J;
val j = _ : void dyn
# val vl = _json j as {name:string, age:int, grade:real} list;
val vl =
  [
   {age = 21,grade = 1.1,name = "Joe"},
   {age = 31,grade = 2.0,name = "Sue"},
   {age = 41,grade = 3.9,name = "Bob"}
  ] : {age: int, grade: real, name: string} list
# val nl = getNames vl;
val nl = ["Joe","Sue","Bob"] : string list
```

Figure 6 Interactive SML# session with a simple example.

```
fun getNames l = map #name l
val j = import J
val vl = _json j as {name:string, age:int, grade:real} list
val nl = getNames vl
```

Figure 6 shows the actual output of an interactive session of this program, where the `JSON` structure has been already opened at the top-level. The SML# compiler type-checks this example and generates the expected results. Some explanations of the compiler output are in order.

- For `getNames`, the compiler infers the polymorphic type `['a#{name:'b}, 'b. 'a list -> 'b list]`.
- The result `j` of `import`ing the JSON string `J` is given the type `void dyn`, which is the representation of type *json* in our implementation.
- The result `vl` of coercing `j` to type `{name:string, age:int, grade:real} list` is a list of records of that type, as expected.
- For the resulting records bound to `vl`, the polymorphic function `getName` is safely applied to yield a list of `string` bound to `nl`.

In writing a JSON string as an ML string constant, control characters such as `"` need to be escaped to `\"`. This is not a big problem, given the fact that JSON objects are usually obtained from external servers or are generated by a program, and there are few occasions to define a JSON object as a string constant. The above artificial example is there for explanation purposes. In writing the JSON string examples below, we omit the out-most `"` and escape characters.

As we discussed in Introduction, collections of JSON objects in typical web services are usually heterogeneous; some fields of each object in the collection may be optional. Suppose

a JSON API requires that a collection of JSON objects have mandatory `name` and `age` fields, and optional `grade` and `nickname` fields. Let J' be the following JSON term compliant with this format:

```
[
 {"name":"Alice", "age":10, "nickname":"Allie"},
 {"name":"Dinah", "age":3, "grade":2.0},
 {"name":"Puppy", "age":7}
]
```

In our scheme, we can type such a heterogeneous collection with partially dynamic records as follows:

```
val j' = import J'
val vl' = _json j' as {name:string, age:int} dyn list
val nl' = getNames (map view vl')
```

While the inferred type of `j'` is `void dyn` as in the previous example, the SML# compiler infers the following typing for `vl'`:

```
val vl' = _ : {name:string, age:int} dyn list
```

This shows that a JSON object containing a heterogeneous collection is typed as a list of partially dynamic records. We note that the record-polymorphic function `getName` can be applied to the resulting partially dynamic records.

When some optional fields are significant, then we can *enlarge* the type of each object separately. The following function picks up either a `nickname` or `name` for each record in the given list depending on the existence of `nickname` field.

```
fun getFriendlyName vl =
    map (fn x => _jsoncase x of {nickname=y:string, ...} => y
                              | _ => #name (view x))
        vl
```

The pattern `{nickname=y:string, ...}` enlarges the partial record type of x to $\{$`nickname` : `string`, ...$\}$ and binds the `nickname` field to y if the enlargement succeeds. This type of enlargement does not affect the type of x. Owing to SML#'s record polymorphism, `getFriendlyName` has a record-polymorphic type

```
['a#{name:string}. 'a dyn list -> string]
```

indicating that this function can be applied to any partial record that has at least a `name` field of `string` type.

5.2 Partial records in the real world

As we briefly mentioned above, collections of partially dynamic records frequently appear in JSON, typically in communication among web services. Examples include the OAuth authorization protocol [13], Twitter search API [28], and Google Maps API Web services [12]. Most other popular web services also provide JSON interfaces.

This subsection demonstrates the benefits of our proposal using a sample in the real world. For this purpose, we choose OpenWeatherMap [23], which is simple but elaborate enough for our purpose of realistic evaluations. OpenWeatherMap provides free access to weather data

over the Internet. JSON is adopted by OpenWeatherMap as a format for sending weather data to the client.

By sending a specific HTTP request to OpenWeatherMap's web server with some parameters, the user can obtain a variety of collections of weather data as an JSON array of JSON objects. Each of the objects consists of fields for weather parameters measured by a weather station such as temperature, wind speed, and amount of precipitation. The concrete form of the weather data may vary for several reasons: the precipitation amount may be absent if it does not rain, the geographic details of cities may be omitted if the user requests data for a particular city, and a record may contain extra system-reserved parameters that are undocumented and would be ignored by clients. Owing to such flexibility of the data structure, the response data is inherently heterogeneous and partial. The following JSON data is an example of the server response consisting of current weather parameters of several cities in Japan:

```
{"list":[
 {"id":2111149, "name":"Sendai-shi", "sys":{···},
  "coord":{"lon":140.87, "lat":38.27}, "weather":[{"main":"Rain",···}],
  "main":{"temp":273.538,···}, "clouds":{"all":92}, "rain":{"3h":1},
  "wind":{"speed":0.45,···}, "dt":1424968138},
 {"id":1857910, "name":"Kyoto", "sys":{···},
  "coord":{"lon":135.75, "lat":35.02}, "weather":[{"main":"Clear",···}],
  "main":{"temp":275.887,···}, "clouds":{"all":8},
  "wind":{"speed":5.21,···}, "dt":1424968420},
 ···],···}
```

In this example, the record of `Sendai-shi` has a `rain` field but `Kyoto` does not, owing to clear weather in `Kyoto`.

A convenient way to read this weather data in SML# is to write the partial record type that specifies only the significant fields. User A, who is interested only in the names and temperatures of cities, would write the following code:

```
_json e as {list:{name:string, main:{temp:real} dyn} dyn list} dyn
```

while another user B, who is interested in other parameters, would write different types of nested partially dynamic records. In our language, the user can control the structure of JSON data to be retrieved in a comfortable, flexible, and type-safe manner. This maximizes the flexibility of programming with JSON objects.

Partially dynamic records and record polymorphism are useful in dealing with this kind of heterogeneous data collection. Consider the following function that calculates the average temperature:

```
fun avgTemp l =
    foldl (op +) 0.0 (map (#temp o view o #main o view) l)
    / real (length l)
```

This function has a record-polymorphic type

```
['a#{main:'b dyn}, 'b#{temp:real}. 'a dyn list -> real]
```

indicating that `avgTemp` is independent of the detail of the structure of weather data. Thus the user A and others who are interested at least in temperature can share this function. Another way to implement this computation is to use `_jsoncase` on dynamic values.

```
fun avgTemp' l =
  foldl (op +) 0.0
    (map (fn x => _jsoncase x of {main={temp:real,...},...} => temp) l)
  / real (length l)
```

This function returns the same value as `avgTemp` but has type `['a. 'a dyn list -> real]`, which is different from that of `avgTemp`. This new typing indicates that `avgTemp'` can be applied to any dynamic data. While `avgTemp'` is more flexible than `avgTemp`, `avgTemp'` may raise a runtime exception owing to dynamic coercion or match failure, whereas `avgTemp` never fails at runtime. The user may choose an appropriate style of programming for dynamic values according to his/her intention.

The flexibility provided by `_jsoncase` is particularly useful when we want to write codes to access an optional field. The following example lists the names of the cities where it rains more heavily than a given threshold.

```
fun rainyCities t cityListInJson =
    List.mapPartial (fn x => if (_jsoncase x of
                                   {rain={"3h"=y:int}, ...} => y > t
                                 | _ => false)
                             then SOME (#name (view x))
                             else NONE)
                    cityListInJson
```

In this example, `{"3h"=···}` is the record pattern that matches with JSON objects consisting of a single `3h` field. Again, this function has a record-polymorphic type, which precisely represents the behavior of the function, similar to the example of `getFriendlyName` in Section 5.1.

Another typical factor that introduces heterogeneousness in practical JSON is number literal. Along with the nature of JavaScript, an integer value (without a fraction part) of JSON may be interpreted as a floating-point value: `5` of JSON usually means either `5` of integers or `5.0` of floating-point numbers. This is also the case for OpenWeatherMap according to its documentation. To deal with this, one would read JSON numbers as `void dyn` as follows:

```
val j = _json e as ···main:{temp:void dyn} dyn···
```

and write the following function that coerces JSON number as `real`:

```
fun getNumAsReal x =
    _jsoncase x of
      (n:int) => Real.fromInt n
    | (r:real) => r
    | _ => 0.0 / 0.0  (* not a number *)
```

We believe that the above examples demonstrate strongly that our JSON extension is beneficial and useful in practical software development with JSON objects including applications with web services.

During the evaluations, we also found a few issues worth considering for further improvements of JSON programming. One is the treatment of JSON numbers. We note that `getNumAsReal` example above is not entirely satisfactory in the spirit of typeful programming since the types of both `j` and `getNumAsReal` are much looser than the user's intention. One possible refinement to our framework is to introduce an additional ad-hoc ordering relation

`int` $\leq$ `real` on JSON types. Another more accurate solution would be to introduce a new type `num` with the ordering relations `int` $\leq$ `num` and `real` $\leq$ `num`. Another possible refinement is to merge `_jsoncase` with ML `case` so that the user can interleave partially dynamic record patterns and ML value patterns. For instance, in the example of `rainyCities`, if the user could write a mixture of the pattern for partially dynamic records `{rain={3h:int}, ...}` and the field selection `#name` at the argument position of `fn`, it would be a more intuitive and convenient shorthand for ML programmers. One approach is to redesign the ML pattern language to integrate JSON patterns and refine the ML pattern matching algorithm with JSON cases. We believe both refinements are feasible, and we would like to investigate them in future.

On the whole, we conclude that our proposal provides a promising basis for the seamless integration of JSON manipulation in ML.

6 Related works

There are a number of libraries for serializing/deserializing JSON objects in statically typed languages such as YoJson [16], Aeson [24], and Atdgen [15]. There are also meta-level tools and frameworks to generate application-specific classes and types for processing JSON objects such as `type_conv` of Camlp4 [8] and `JsonProvider` of F# 3.0 [27]. In these approaches, however, JSON objects and their associated functions are not part of the type system of the underlying programming languages. As a result, they do not achieve type-safe and seamless programming with JSON objects.

In the general perspective of developing typing discipline for implicitly typed data formats, our work is related to a number of works that investigate type structures of XML and RDF. XDuce and CDuce type systems [14, 3] represent static structures of XML based on regular expression types. Frisch et al. [11] investigates semantic subtyping as a basis for CDuce-like type systems. These type systems are more powerful than ours, and a number of features such recursive structures and (tag-less) unions are cleanly represented. Perhaps owing to their flexibility, however, regular expression types are not related to static data structures in a type system of a polymorphic programming language. As we have mentioned in Introduction, our goal and contribution is to represent JSON structures using labeled record types in ML so that they can be directly manipulated in ML using polymorphic record operations.

Regarding JSON objects, Colazzo et al. [7] recently presented a type inference algorithm for large JSON data-sets using union types and subtyping. Compared with the partial record types we have used, their approach could potentially find more accurate types for a large collection of heterogeneous JSON objects. It is, however, not immediately obvious whether their JSON type system can be integrated in a static type system of a programming language.

Our approach is based on partially dynamic records [6], whose central idea is to represent a type of a heterogeneous collection as a record structure common to all components in the collection. From this perspective, it is related to gradual typing proposed by Siek and Taha [25]. Their approach uses subtyping and dynamic typing. This combination can represent dynamically typed heterogeneous collections in a statically typed language. Largely based on this observation, Bierman et al. [4] gave a formal account for the core of TypeScript. This approach is also adopted in Flow [9]. As we commented in the Introduction, however, subtyping would complicate polymorphism and type inference, and its impact on the implementation method remains to be investigated. Kiselyov et al. [19] presented an implementation technique to encode heterogeneous lists by exploiting Haskell type classes and their extensions including multi-parameter classes and functional dependencies. Its type

theoretical property is, however, not well investigated, and its type theoretical properties and their relationship to ours are not clear.

7 Conclusions and further investigation

JSON is a widely accepted format for data exchange over the Internet, especially among web services. In this paper, we have developed a typed programming language for seamless and high-level manipulation of JSON data. The major obstacle to the seamless introduction of JSON manipulation to a typed language is the heterogeneous nature of JSON. We have presented a typed calculus to deal with heterogeneous JSON objects based on partially dynamic records, and have established its type soundness. The proposed calculus has been fully implemented as an extension to the SML# compiler. In the extended language, the programmer can directly import a JSON object as a term of static data structure composed of labeled records and lists with the full benefits of static polymorphic type-cheking. The implementation also provides `_jsoncase` syntax for ML-style pattern matching. We have demonstrated the feasibility and the benefits of our approach through examples that interact with actual web services.

The important work we are now planning to conduct is to put our proposed language into serious software development in industry, and to evaluate its benefits in software development. From a more technical perspective, we plan to investigate programming language support for manipulating JSON-based databases. We would like to have a declarative query language for a large and complex JSON data, and would like to integrate it into the SML# compiler. A possible approach toward such an integration is to extend the type system proposed in this paper and the integration of SQL in a programming language with the record polymorphism we reported in [21]. Another interesting future work is to extend our technique to data formats similar to JSON, such as RDF and XML.

Acknowledgments

The authors thank the members of the project “Research and Development on Highly Functional and Highly Available Information Storage Technology” sponsored by the Ministry of Education, Culture, Sports, Science and Technology in Japan for discussions.

The authors also thank the anonymous reviewers for careful reading of the paper and a number of constructive comments.

References

1 M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

2 M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *ACM Transcations on Programming Languages and Systems*, 13(2):237–268, 1991.

3 V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of the ACM International Conference on Functional Programming*, pages 51–63, 2003.

4 G. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In *Proceedings of the European conference on Object-Oriented Programming*, pages 257–281. Springer Berlin Heidelberg, 2014.

5 T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159 (Proposed Standard), 2014. URL: `http://www.ietf.org/rfc/rfc7159.txt`.

6 P. Buneman and A. Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–74, 1996.

7 D. Colazzo, G. Ghelli, and C. Sartiani. Typing massive JSON datasets. In *Proceedings of the International Workshop on Cross-model Language Design and Implementation (XLDI)*, Copenhagen, Denmark, 2012.

8 J. Dimino. Camlp4, 2014. URL: `https://opam.ocaml.org/packages/camlp4/camlp4.4.03.0/`.

9 Flow | a static typechecker for JavaScript. URL: `http://flowtype.org`.

10 A. Frisch. OCaml + XDuce. In *Proceedings of the ACM International Conference on Functional Programming*, pages 192–200, 2006.

11 A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4), 2008.

12 Google Maps APIs | Google Developers. URL: `https://developers.google.com/maps/`.

13 D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, 2012. URL: `http://www.ietf.org/rfc/rfc6749.txt`.

14 H. Hosoya and B. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Technol.*, 3(2):117–148, 2003.

15 M. Jambon. Atdgen, 2010. URL: `https://github.com/mjambon/atdgen`.

16 M. Jambon. Yojson: JSON library for OCaml, 2010-2012. URL: `https://github.com/mjambon/yojson`.

17 JSON schema. URL: `http://json-schema.org`.

18 G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer Verlag, 1987.

19 O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 96–107, 2004.

20 A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995. A preliminary summary appeared at ACM POPL, 1992 under the title "A compilation method for ML-style polymorphic record calculi.".

21 A Ohori and K. Ueno. Making Standard ML a practical database programming language. In *Proceedings of the ACM International Conference on Functional Programming*, pages 307–319, 2011.

22 A. Ohori, K. Ueno, K. Hoshi, S. Nozaki, T. Sato, T. Makabe, and Y. Ito. SML# in industry: A practical ERP system development. In *Proceedings of the ACM International Conference on Functional Programming*, pages 167–173, 2014.

23 OpenWeatherMap.org. OpenWeatherMap, 2012-2016. URL: `http://openweathermap.org`.

24 B. O'Sullivan. Aeson: Fast JSON parsing and encoding, 2011-2014. URL: `https://hackage.haskell.org/package/aeson`.

25 J. Siek and W. Taha. Gradual typing for objects. In *Proceedings of the European conference on Object-Oriented Programming*, pages 2–27, 2007.

26 SML#, 2006-2016. URL: `http://www.riec.tohoku.ac.jp/smlsharp/`.

27 D. Syme, K. Battocchi, K. Takeda, D. Malayeri, J. Fisher, J. Hu, T. Liu, B. McNamara, D. Quirk, M. Taveggia, W. Chae, U. Matsveyeu, and T. Petricek. F#3.0 - strongly-typed language support for internet-scale information sources. Technical Report MSR-TR-2012-101, Microsoft Research, 2012.

28 The Search API | Twitter Developers. URL: `https://dev.twitter.com/rest/public/search`.

Higher-Order Demand-Driven Program Analysis

Zachary Palmer[1] and Scott F. Smith[2]

1 **Swarthmore College**
Swarthmore, PA, USA
zachary.palmer@swarthmore.edu
2 **The Johns Hopkins University**
Baltimore, MD, USA
scott@cs.jhu.edu

Abstract

We explore a novel approach to higher-order program analysis that brings ideas of on-demand lookup from first-order CFL-reachability program analyses to higher-order programs. The analysis needs to produce only a control-flow graph; it can derive all other information including values of variables directly from the graph. Several challenges had to be overcome, including how to build the control-flow graph on-the-fly and how to deal with non-local variables in functions. The resulting analysis is flow- and context-sensitive with a provable polynomial-time bound. The analysis is formalized and proved correct and terminating, and an initial implementation is described.

1998 ACM Subject Classification F.3.2. Semantics of Programming Languages

Keywords and phrases functional programming, program analysis, polynomial-time, demand-driven, flow-sensitive, context-sensitive

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.19

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.2.1.9

1 Introduction

Flow analysis for imperative first-order programs is a well-known and straightforward process. Before the analysis starts, it is known which functions are being invoked at each call site in the source program. This is possible because, in absence of higher-order functions, potential control flow is determined immediately from the structure of the program. So, for first-order programs, it is possible to directly build a fixed control flow graph (CFG) where each edge in the CFG points to a potential next program point. The program analysis then can monotonically accumulate information about what values program variables could take on at each program point, propagating information along this fixed CFG.

Forward higher-order program analysis

In languages with higher-order functions, program analysis is much more challenging: it is no longer obvious which functions may be invoked at each call site. The program's data flow determines which functions appear at the call site, in turn influencing the program's control flow. Accurate analyses of programs with higher-order functions must therefore compute data- and control-flow information simultaneously [12, 16].

Higher-order program analyses are generally based on abstract interpretations [2]; such analyses define a finite-state abstraction of the operational semantics transition relation to

30th European Conference on Object-Oriented Programming (ECOOP 2016).
Editors: Shriram Krishnamurthi and Benjamin S. Lerner; Article No. 19; pp. 19:1–19:25


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

soundly approximate the program's runtime behavior. The resulting analysis has the same general structure as the operational semantics it was based on: program points, environments, stacks, stores, and addresses are replaced with abstract counterparts which have finite cardinality, "hobbling" the full operational semantics of the language to guarantee termination of the analysis [13]. A sound analysis will visit the (finitely many) abstract counterparts of all reachable concrete program states, producing a finite automaton representing all potential program runs.

We use the term "forward analyses" to refer to standard higher-order program analyses, to emphasize how they propagate data forward through the program in the same manner as an operational semantics does. All abstract interpretation based higher-order program analyses, including [12, 16, 22, 13, 27, 8, 15, 3], are forward analyses.

Since each node is an abstract state and there can be a great many combinations of data for environment or store, even the finitized abstract state space can be very large. For this reason, any practical analysis must compress the total number of states. Typically, the program counter is preserved. One standard compression, *store widening*, replaces the store in each each abstract program state with a single global store; this global store is then the union of all the individual stores.

Store widening is effective in compressing the abstract state set but tends to give too little precision. For example, function polymorphism which is present before store widening may be lost as the parameters to the calls of a given function in each program state are unioned together. Store widening also generally loses any so-called flow sensitivity, where a given (often stateful) variable can take on different values at different points.

So, other methods are employed to recover expressiveness without a full store in each abstract state. One method is polyvariance, such as in kCFA [22] for $k > 0$, where each function parameter gets unique addresses relative to the k most recent stack frames. Another method is call-return alignment: the analysis uses a stack internally to return only to the call site of the particular call being analyzed [26]. Call-return alignment also gives some, but not all, of the power of polyvariance. One alternative to store widening is abstract garbage collection [14], which also limits the number of abstract states: the store is not widened, but an analysis-time analogue of run-time garbage collection removes inaccessible bindings from the store. Abstract GC preserves flow-sensitivity, but this technique is still susceptible to an explosion in analysis states for programs with nontrivial amounts of persistent data as each configuration of this data represents another abstract program state to consider.

A demand-driven approach to higher-order programs

In this paper, we explore a fundamentally different approach to higher-order program analysis which does not proceed by finitizing an operational semantics. Rather than pushing data forward as in an operational semantics, the analysis looks up data on demand. This lookup walks backward along the control flow from the point at which a variable's value is needed to the point at which the variable was most recently defined. Actual values *never* need to be pushed forward so, in some sense, lookup is lazier than a standard lazy interpreter.

CFL-reachability analyses [6] have shown how demand-driven variable lookup is possible in first-order program analyses. In comparison to forward analyses, the demand-driven approach has potential for better performance: the aforementioned optimizations are unnecessary as no store is propagated forward through the analysis. CFL-reachability is a first-order analysis, where the CFG is already known and where there are no non-local variables in functions. Demand-driven lookup is applied to higher-order languages in [18], but in the a context of a flow-insensitive type-based analysis with other restrictions. To

generally apply this technique to higher-order programs, we must have (1) a technique for constructing the CFG in tandem with the data flow analysis and (2) a strategy for handling non-local variables.

In this paper, we solve these two problems to produce a higher-order Demand-Driven Program Analysis (DDPA). The CFG is constructed incrementally just as in a forward higher-order program analysis. Unlike a forward analysis, we *only* need to construct the CFG; there is *no* abstracted environment/store/etc as all of the variable lookup happens in reverse by following CFG edges backwards. To deal with non-local variables in functions, we develop a novel reverse lookup process that is related to how access links are used in a compiler to look up non-local variables: we find the definition of the function itself and look up the variable from there. For added precision, we incorporate call-return alignment from CFL-reachability [21, 20] and pushdown higher-order flow analyses [26, 8].

Contrast with forward analyses

During the course of DDPA, each variable lookup occurs relative to a particular point in the (partial) control flow graph where the variable is used; this makes flow-sensitivity a natural component of DDPA. In forward higher-order analyses, such flow-sensitivity is expensive since it prohibits common performance-enhancing techniques such as store widening. Abstract garbage collection, an aforementioned alternative to store widening, prunes the stores significantly but produces many states for persistent data. Additionally, there is no need for explicit polyvariance in DDPA: the combination of call-return alignment and non-local lookup achieves the full effect of polyvariance without an explicit polyvariance model *a la* kCFA. DDPA appears to compare favorably to forward analyses: it has the simultaneous benefits of store widening (we have no per-node store), abstract garbage collection (demand-driven lookup prevents the generation of garbage), call-return alignment (the one feature we directly adopt) and polyvariance (which is subsumed by our call-return alignment given our novel non-local variable lookup mechanism). Forward analyses cannot combine all these features because abstract GC is ineffective in the presence of store widening [8].

In theory, DDPA has minimal requirements on run-time data structures: the only required data structure is the CFG, which is minuscule compared to the formal treatment of forward analysis. In practice, an efficient DDPA implementation requires significant caching structures which are similar to fragments of an abstract store in a forward analysis. We do not assert DDPA to be a strictly better form of program analysis, only a substantially *different* one with different trade-offs and worthy of study.

In terms of technical overhead, DDPA has the advantage of naturally scaling up to a flow-, path-, and context-sensitive analysis while preserving a relatively compact formal specification. On the other hand, reverse lookup is a different process than the original (forward) operational semantics and requires a major re-thinking about program flow; a forward analysis can to some degree be "read off the operational semantics" [25, 13].

We have completed a proof-of-concept implementation of DDPA, described in Section 6, and include it as an artifact archived with the paper. The implementation builds a pushdown automaton (PDA) and variable lookup questions can be cast as reachability questions in the PDA. This implementation has not been optimized but it confirms that the analysis has the expected behavior on examples.

e	::=	$[c, \ldots]$	*expressions*	v	::=	$r \mid f$	*values*
c	::=	$x = b$	*clauses*	r	::=	$\{\ell, \ldots\}$	*records*
b	::=	$v \mid x \mid x\ x \mid x \sim p\ ?\ f : f$	*clause bodies*	f	::=	`fun` x `->` `(` e `)`	*functions*
x			*variables*	p	::=	r	*patterns*
E	::=	$[x = v, \ldots]$	*environments*				

Figure 1 Expression Grammar.

Paper outline

Section 2 presents an example-driven description of the main features of the analysis. Section 3 gives the full details, and Section 4 establishes its soundness relative to an operational semantics. Section 5 defines extensions for full records, path-sensitivity, and state. Section 6 describes the implementation. Section 7 covers related work. We conclude in Section 8.

2 Overview of the Analysis

This section informally presents DDPA by example.

2.1 A Simple Language

For a given program, our objective is to establish the possible execution paths that the program will take. We encode this information in the form of a CFG, more precisely as a *happens-before* relation $c \ll c'$: program point c related to c' means there may be control flow from c to c'.

We use a simple functional language defined in Figure 1. To make it easier to keep track of program points and operation sequencing, we restrict our language syntax to a shallow A-normal form (ANF) [5]. We further restrict all variables to be unique; this ensures that clauses c themselves denote program points. (**Notation**: lists are written $[g_1, \ldots, g_n]$, and $||$ denotes list concatenation. We may overload as $g_1 \,||[g_2, g_3] = [g_1, g_2, g_3]$ when it is unambiguous.)

The operational semantics of this (eager) language are straightforward and are given in Section 4.1. Note that we have a degenerate form of record with label components only; this restriction is made to simplify the formalism and we relax this restriction in the implementation. Case analysis is written $x \sim p\ ?\ f_1 : f_2$; we execute f_1 with x as argument if the value of x matches the pattern p, and f_2 with x as argument otherwise.

2.2 The Basic Analysis

Consider the program in Figure 2. Note that `f` is just a fancy identity function used to help illustrate the analysis, and that clause `n0` is a "no-op": it is present only to help clarify the diagrams.

The analysis begins by constraining the top-level clauses to happen in order: `f`, `x1`, `z1`, `x2`, and then `z2`. For technical reasons, we also add distinguished Start and End nodes.

Figure 3 shows the completed analysis; up to now we have only described the bottom row. Focusing only on that row for now, the solid arrows ($\longrightarrow$) represent nodes ordered by $\ll$ in the source program, and indicate what we know about control flow at the beginning of the analysis: only that control can move from each clause to the next. As shorthand, we

```
1  f = fun x -> ( # λx.(λy.y)x
2      i = fun a -> (
3          r2 = a
4      );
5      r = i x;
6      n0 = r;
7  );
8  x1 = {y};
9  z1 = f x1;      # evaluates to {y}
10 x2 = {n};
11 z2 = f x2;      # evaluates to {n}
```

Figure 2 Invocation Example: ANF.

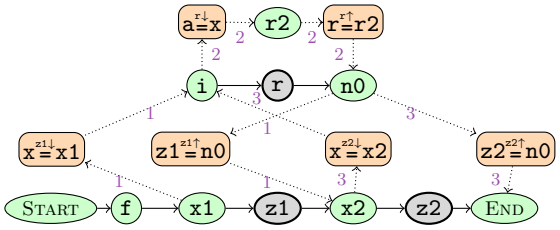


Figure 3 Invocation Example: Analysis.

identify each clause by the unique variable it assigns; thus, the green node labeled `x1` refers to line 8 in Figure 2. Green nodes are immediate clauses and gray nodes are call sites.

2.2.1 Adding control flows for function calls

To elaborate the call graph, we must connect each call site to any function body invoked at that call site. Proceeding forward on the control flow from the program start, the first call site (gray node) is node `z1`. The dotted edges (⇢) annotated with the number "1" represent control passing from call site `z1` to the start of the body of `f` (the `i`, `r`, `n0` clauses), and from the end of the function body back to `z1`. Note that the wires in fact run "around" the call site in the figure, going out the predecessor of `z1` and back to the successor. This reflects how a function inlining would work. (Aside: both ⟶ and ⇢ edges define $\ll$ relationships between program points; the two edge sorts are used for illustration only.)

Additionally, the call adds two *new* intermediate nodes: `x=x1` represents copying the argument `x1` at the call site into the function `f`'s parameter, while `z1=n0` represents copying the result of the function into the variable defined at the call site. These nodes are marked with annotations to indicate their call site and purpose; `z1`↓, for instance, indicates a parameter passed in from the `z1` call site. All of the information in these new nodes can in fact be inferred from context since all values are unique given a (call site, function) pair. In other words, the happens-before relation is isomorphic to the call graph.

Continuing with the analysis: now that the body of `f` has been wired in, the call site `r` can execute, adding two orange wiring nodes and the corresponding "2" flows. The "3" flows are similarly added when we elaborate `z2`.

2.2.2 Variable lookup as reverse walk

In the description above, we glossed over how variable lookup occurs: for each call site, we needed to look up the particular functions to invoke and this can be nontrivial in a higher-order language. The clause `z1` invokes `f`, and all potential definitions of `f` need to be wired to the call site. This particular case is simple since `f` obviously has only one value.

In general, variable lookup is *contextual*: lookup starts at the clause where the variable is being used and proceeds backwards in time with respect to the control flow to find the most recent value. This approach to lookup is related to the demand-driven optimizations of first-order CFL-reachability analyses [6].

For a less trivial example, let us look up `x` from the perspective of `r`'s defining clause in line 5. Walking back though the graph from `r`, there are two definitions of `x` reached:

x=x1 and x=x2. Since x1/x2 are not concrete values, lookup proceeds by looking up those variables, respectively. Ultimately, x has final value set {{y}, {n}}, the set of arguments the function was invoked on.

Since variable lookup always starts at the node representing the redex where the variable is used, we obtain a flow-sensitive analysis by default. Note that the specification of lookup traverses all the way back to the original definition every time; for example, in looking up x we needed to look up x=x1 and we then needed to continue back to x1's original definition. An efficient implementation should cache, but that is orthogonal to the specification.

2.3 Constraining lookup to reasonable call stacks

The above description of a simple lookup leaves out an important refinement in DDPA: it is possible to rule out variable lookup search paths on which calls and returns do not align, because such a path corresponds to no program execution. To show the incompleteness of lookup as described thus far, consider how lookup would find values for z2 from the perspective of the end of the program. To find z2, we proceed back on the control flow to z2=n0 (recall how the call site itself is dead code after all calls have been wired in), at which point the search is now for the value of n0; continuing back we reach n0=r and proceed by looking up r from node n0, and so on until we are looking up x from node i. Here, there are two paths, to either x=x1 or x=x2. So, we take both paths and union the result, obtaining {{y}, {n}}. This is clearly a loss of precision: {y} cannot appear as the argument of call site z2 at runtime. But, looking at the overall path we took above in this spurious lookup, we walked back into the function via the z2 call site and came out of the front of the function to the z1 call site. This is a non-sensical program execution path since the call and return site are not aligned.

The annotations z1↓ / z1↑ on the wiring nodes indicate a call into and return from call site z1, respectively, and are used to filter out these spurious paths. On any lookup path the wiring annotations must pair correctly; that is, one may only visit a z1↓ node if a corresponding z1↑ node was visited. We track these pairings using an abstract call stack to verify they obey a reasonable call-return discipline. The idea of call-return alignment is taken from CFL-reachability [21, 20] and higher-order pushdown analyses [26, 8].

Note that the above analysis exhibits polyvariant behavior: different invocations of the same function are not analyzed uniformly. Polyvariance is commonly achieved by copying, kCFA being a canonical example [16]. In DDPA, polyvariance can be achieved solely by call-return alignment, unlike [26, 8] – stack alignment in these analyses only aligns variables local to the function body, whereas DDPA also aligns non-locals; the subtleties of non-local lookup are covered next.

2.4 Looking up Non-Local Variables

So far, our examples of lookup have been restricted to local variables. However, the above lookup process does not accurately reflect how non-local variables are bound: if non-local variables were handled in the same way, lookup would give non-locals a dynamically scoped semantics. To see why, consider the code in Figure 4 and its graph in Figure 5. If we look up the value of z from the end of the program, we find ourselves asking for the values of v at the wiring of j=e. Proceeding as described above, j does not match v and we would attempt to find a value for v at the e clause. This leads us to the clause v=b, which in turn gives us *just* the value {b}. This would be unsound as it is not consistent with run-time behavior.

```
1 k = fun v -> ( # λx.λy.x
2     k0 = fun j -> ( r = v; );
3 );
4 a = {a};
5 f = k a;          # λy.{a}
6 b = {b};
7 g = k b;          # λy.{b}
8 e = {};
9 z = f e;          # evaluates to {a}
```

Figure 4 Non-Local Variable: ANF.

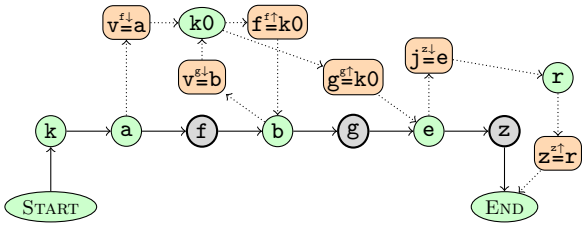


Figure 5 Non-Local Variable: Analysis.

To give lexical scoping semantics, the lookup function must be modified. Lexical scoping means we want the values of non-local variables at the point which the function containing them was *itself* declared, so we proceed by first locating the function definition and then resuming our search for the value of the variable. This is similar to the access link method of non-local lookup in a compiler implementation. In the example above, the pivotal decision is made when finding values of `v` when we have searched through the whole body of the function in lines 2-4 and arrived at the argument wiring `j=e`. From the annotation `z↓` on this node, we can deduce that we are leaving the current function which was called from site `z`. Since we have not yet found `v`, it must be a non-local. So, we then delay our search for `v`, examine the call site `z` (appearing in the wiring annotation), and look up the *function* invoked at that point; here, this means to search for `f` starting from `e`. This search leads us through `f=k0` to `k0`, the point at which the function was originally defined, and from here the search for `v` can soundly resume. Using this approach, the lookup of `z` from the end of the program yields `{a}`.

The general case of non-local lookup chains on the above idea, since the defining function could itself be non-locally defined. The chain is up the lexical scoping structure, so its length is bound by program size.

2.5 Looking up Higher-Order Functions

The analysis described thus far also does not deal well with the case of multiple functions showing up at a call site – it does not take in account which functions are actually available. Consider the code example in Figure 6.

We would like a search for `rb` from the End to conclude {`{b}`}, the run-time result. In the process of trying to answer that question, the analysis eventually gets to node `n0`, querying for the variable `hr`. At that point, there are two paths by which to proceed: one to `hr=a` and another to `hr=b`. If the analysis took both paths, the result would be {`{a}`, `{b}`}, but `{a}` will never arise at runtime.

We address this case as follows. Before entering a function call (in reverse), the analysis first runs a subordinate lookup for all functions that could show up at the call site under consideration. The main lookup can then use this result to rule out any function definition that could have not reached the call site in the current context. In the given example, at the node `n0`, before entering node `hr=a` or `hr=b`, the analysis performs a subordinate lookup for `f` from `hr`, under the current calling context of `rb`. In order to align with the `rb=n0` entrance node, the only possible answer is `fb`, ruling out the choice of `hr=a` and leading to a final answer of only `{b}`.

```
h = fun f -> ( # λf. f {}
    e = {};
    hr = f e;
    n0 = hr;
);
fa = fun ia -> ( a = {a}; ); # λ_. {a}
fb = fun ib -> ( b = {b}; ); # λ_. {b}
ra = h fa;
n1 = {};
rb = h fb;          # evaluates to {b}
```

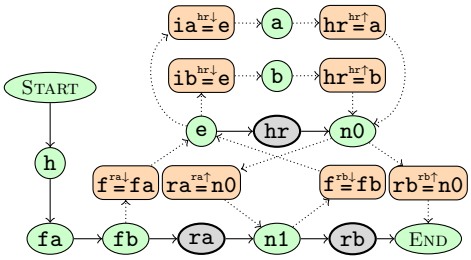


Figure 6 Higher-Order Functions: ANF.

Figure 7 Higher-Order Functions: Analysis.

```
f = fun s -> ( # l-projecting function
    f0 = fun a -> (
        n0 = {};
        r = a ~ {l}
          ? fun a1 -> ( n2 = {};
                ss = s s;  # self-apply
                v = a1.l;  # project l
                r1 = ss v; # recurse
                n3 = r1; )
          : fun a2 -> ( r2 = a2; );
        n1 = r;
    );
);
ff = f f;      # initial self-application
x1 = {};
x2 = {l=x1};
x3 = {l=x2};   # the record {l={l={}}}
z = ff x3;     # evaluates to {}
```

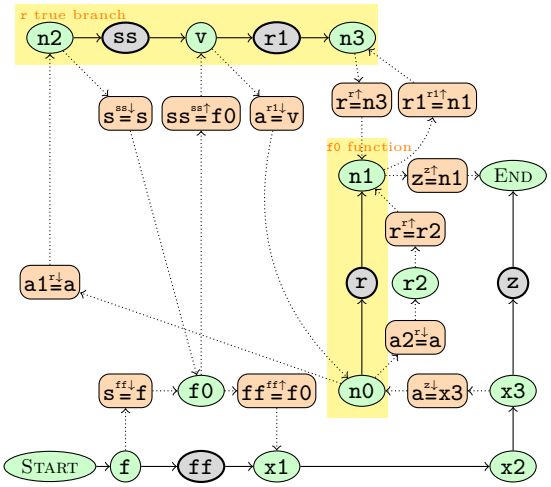


Figure 8 Recursion: ANF.

Figure 9 Recursion: Analysis.

2.6 Recursion and Decidability

The analysis described above naturally analyzes recursive programs: consider the program in Figure 8 which we code in an extension including proper records. This program defines a function `f` which is recursive (by self-passing) and which deeply projects any number of `l` fields from a record.

The DDPA result appears in Figure 9. Lookup generally proceeds via the same process as for the previous examples. The only complication with recursion is that some control flow paths could be cyclic. For instance, the path `n0` $\rightsquigarrow$ `n2` $\rightsquigarrow$ `v` $\rightsquigarrow$ `n0` is a cycle representing an arbitrary number of recursive unwrappings, each of which pushes `r`$\downarrow$ and `r1`$\downarrow$ onto the stack. The specification of the analysis only requires variable lookup paths to be balanced in calls and returns, so there are arbitrarily many possible paths. It is clear in this example that the number of times around the cycle can be bounded without changing the result; as we will discuss later, it is not known whether lookup over an arbitrary-size call stack is computable. We address this issue in Section 3 by defining a stack finitization kDDPA which retains the k most recent contexts in the spirit of kCFA [22].

DDPA is flow-sensitive and context-sensitive. However, it is not path-sensitive; that is, a variable's values are not considered in terms of how branching decisions at conditionals led to a particular program point. Consider for example the values of `z` which may appear

$$
\begin{array}{lcll}
\hat{e} & ::= & [\hat{c}, \ldots] & \textit{abstract expressions} \\
\hat{c} & ::= & \hat{x} \texttt{ = } \hat{b} & \textit{abstract clauses} \\
\hat{b} & ::= & \hat{v} \mid \hat{x} \mid \hat{x}\ \hat{x} \mid & \textit{abs. bodies} \\
 & & \hat{x} \texttt{ ~ } \hat{p} \texttt{ ? } \hat{f} \texttt{ : } \hat{f} & \\
\hat{x} & ::= & x & \textit{abstract variables} \\
\hat{v} & ::= & \hat{r} \mid \hat{f} & \textit{abstract values} \\
\hat{p} & ::= & \hat{r} & \textit{abstract patterns} \\
\hat{f} & ::= & \texttt{fun } \hat{x} \texttt{ -> (} \hat{e} \texttt{)} & \textit{abs. functions} \\
\hat{r} & ::= & r & \textit{abstract records}
\end{array}
\qquad
\begin{array}{lcll}
\hat{V} & ::= & \{\hat{v}, \ldots\} & \textit{abstract value sets} \\
\hat{a} & ::= & \hat{c} \mid & \textit{abs. annotated clauses} \\
 & & \hat{x} \overset{\hat{c}\downarrow}{=} \hat{x} \mid \hat{x} \overset{\hat{c}\uparrow}{=} \hat{x} \mid \textsc{Start} \mid \textsc{End} & \\
\hat{d} & ::= & \hat{a} \ll \hat{a} & \textit{abstract dependencies} \\
\hat{D} & ::= & \{\hat{d}, \ldots\} & \textit{abs. dependency graphs} \\
\hat{X} & ::= & [\hat{x}, \ldots] & \textit{abs. var lookup stacks}
\end{array}
$$

Figure 10 Analysis Grammar.

at the end of the program of Figure 8: the path `x3` $\rightsquigarrow$ `n0` $\rightsquigarrow$ `r2` $\rightsquigarrow$ `n1` $\rightsquigarrow$ End in Figure 9 is valid, but this would imply that the value `{l={l={}}}` is possible for `z`, but it will not arise at runtime. Path sensitivity can close this gap, and we sketch such an extension in Section 5.2. Note this example also uses "real" records and not just the label sets of our grammar of Figure 1; we sketch an extension to full records in Section 5.1.

3 The Analysis Formally

In this section we formalize the analysis algorithm. The operational semantics is standard and so we postpone it and the soundness proof to Section 4.

The grammar constructs needed for the analysis appear in Figure 10. The items on the left are just the hatted versions of the corresponding program syntax. As mentioned above, we require variables to be bound uniquely so that each program point is defined by exactly one variable. Furthermore, we assume the expressions we analyze to be closed: a variable is not used until after the clause in which it is bound.

Edges in a dependency graph $\hat{D}$ are dependencies $\hat{d}$, are written $\hat{a} \ll \hat{a}'$ and mean clause $\hat{a}$ happens right before clause $\hat{a}'$. New clause annotations $\hat{c}\downarrow$ / $\hat{c}\uparrow$ are used to mark the entry and exit points for functions/cases.

▶ **Definition 3.1.** We use the following notational sugar for graph dependencies:

- We write $\hat{a}_1 \ll \hat{a}_2 \ll \ldots \ll \hat{a}_n$ to mean $\{\hat{a}_1 \ll \hat{a}_2, \ldots, \hat{a}_{n-1} \ll \hat{a}_n\}$.
- We write $\hat{a}' \ll \{\hat{a}_1, \ldots, \hat{a}_n\}$ (resp. $\{\hat{a}_1, \ldots, \hat{a}_n\} \ll \hat{a}'$) to denote $\{\hat{a}' \ll \hat{a}_1, \ldots, \hat{a}' \ll \hat{a}_n\}$ (resp. $\{\hat{a}_1 \ll \hat{a}', \ldots, \hat{a}_n \ll \hat{a}'\}$).
- We write $\hat{a} \lessdot \hat{a}'$ to mean $\hat{a} \ll \hat{a}' \in \hat{D}$ for some graph $\hat{D}$ understood from context.
- We define abbreviations $\textsc{Preds}(\hat{a}) = \{\hat{a}' \mid \hat{a}' \lessdot \hat{a}\}$ and $\textsc{Succs}(\hat{a}) = \{\hat{a}' \mid \hat{a} \lessdot \hat{a}'\}$.

▶ **Definition 3.2.** Initial embedding $\textsc{Embed}([c_1, \ldots, c_n])$ is the graph $\hat{D}$ given by $\textsc{Start} \ll \hat{c}_1 \ll \ldots \ll \hat{c}_n \ll \textsc{End}$, where each $\hat{c}_i = c_i$.

This initial graph is just the linear sequence of clauses in the "main program".

3.1 Lookup

As was described in Section 2, the analysis will look back along $\ll$ edges in the graph $\hat{D}$ to search for definitions of variables it needs. We now define this lookup function.

3.1.1 Context Stacks

The definition of lookup proceeds with respect to a current *context stack* $\hat{C}$. The context stack is used to align calls and returns to rule out cases of looking up a variable based on a non-sensical call stack, and was described in Section 2.3.

The proof of decidability relies upon bounding the depth of the call stack. We first define a general call stack model for DDPA, and in Section 3.3 below we instantiate the general model with a fixed k-depth call stack version notated kDDPA; this is a simple bounding strategy and our model can in principle work with other strategies.

▶ **Definition 3.3.** A *context stack model* $\Sigma = \langle \hat{\boldsymbol{C}}, \epsilon, \textsc{Push}, \textsc{Pop}, \textsc{IsTop} \rangle$ obeys the following:

1. $\hat{\boldsymbol{C}}$ is a set. We use $\hat{C}$ to range over elements of $\hat{\boldsymbol{C}}$ and refer to such $\hat{C}$ as *context stacks*.
2. $\epsilon \in \hat{\boldsymbol{C}}$.
3. $\textsc{Push}(\hat{c}, \hat{C})$ and $\textsc{Pop}(\hat{C})$ are total functions returning stacks.
4. $\textsc{IsTop}(\hat{c}, \textsc{Push}(\hat{c}, \hat{C}))$, $\textsc{IsTop}(\hat{c}, \epsilon)$, and $\textsc{Pop}(\epsilon) = \epsilon$ hold.
5. If $\textsc{IsTop}(\hat{c}, \hat{C})$ then $\textsc{IsTop}(\hat{c}, \textsc{Pop}(\textsc{Push}(\hat{c}', \hat{C})))$.

The context stack represents the calls of which we are *certain*; thus, the empty stack represents "no knowledge" (and not "no stack frames"). Thus, popping from the empty stack yields the empty stack and any clause is considered to be on top of an empty stack.

3.1.2 Lookup stacks

Lookup also proceeds with respect to a *non-locals lookup stack* $\hat{X}$ which is used to remember non-local variable(s) we are in the process of looking up while searching for the lexically enclosing context where they were defined. In the general case it is a stack since the function itself could prove to be a non-local, etc. This stack can be unbounded unlike the context stack above. Section 2.4 gave motivation and examples for non-local variable lookup.

3.1.3 Defining the lookup function

Lookup finds the value of a variable starting from a given graph node. Given a dependency graph $\hat{D}$, we write $\hat{D}(\hat{x}, \hat{a}_0, \hat{X}, \hat{C})$ to denote the lookup of a definition of the variable $\hat{x}$ in $\hat{D}$ relative to graph node $\hat{a}_0$, access stack $\hat{X}$ and context stack $\hat{C}$. We let $\hat{D}(\hat{x}, \hat{a}_0)$, the top-level lookup of $\hat{x}$ from graph node $\hat{a}_0$, abbreviate $\hat{D}(\hat{x}, \hat{a}_0, [], \epsilon)$. Note that we have oriented the definition so looking from graph node $\hat{a}_0$ means we are not looking for the value in that node itself, but in (all) predecessors of it.

▶ **Definition 3.4.** Given dependency graph $\hat{D}$, $\hat{D}(\hat{x}, \hat{a}_0, \hat{X}, \hat{C})$ is the function returning the least set of values $\hat{V}$ satisfying the following conditions:

1. If $\hat{a}_1 = (\hat{x} = \hat{v})$, $\hat{a}_1 \lll \hat{a}_0$, and $\hat{X} = []$, then $\hat{v} \in \hat{V}$.
2. If $\hat{a}_1 = (\hat{x} = \hat{f})$, $\hat{a}_1 \lll \hat{a}_0$ and $\hat{X} = [\hat{x}_1, \ldots, \hat{x}_n]$ for $n > 0$, then $\hat{D}(\hat{x}_1, \hat{a}_1, [\hat{x}_2, \ldots, \hat{x}_n], \hat{C}) \subseteq \hat{V}$.
3. If $\hat{a}_1 = (\hat{x} = \hat{x}')$ and $\hat{a}_1 \lll \hat{a}_0$, then $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$.

4a. If $\hat{a}_1 = (\hat{x} \overset{\hat{c}\downarrow}{=} \hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$, $\hat{c}$ is an application clause, and $\textsc{IsTop}(\hat{c}, \hat{C})$, then $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \textsc{Pop}(\hat{C})) \subseteq \hat{V}$.

4b. If $\hat{a}_1 = (\hat{x} \overset{\hat{c}\downarrow}{=} \hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$ and $\hat{c}$ is a conditional clause, then $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$.

5a. If $\hat{a}_1 = (\hat{x} \overset{\hat{c}\uparrow}{=} \hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$ and $\hat{c} = (\hat{x}_r = \hat{x}_f\ \hat{x}_v)$, then $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \textsc{Push}(\hat{c}, \hat{C})) \subseteq \hat{V}$, provided $\texttt{fun}\ \hat{x}'' \texttt{->} (\hat{e}) \in \hat{D}(\hat{x}_f, \hat{c}, [], \hat{C})$ and $\hat{x}' = \textsc{RV}(\hat{e})$.

5b. If $\hat{a}_1 = (\hat{x} \overset{\hat{c}\uparrow}{=} \hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$ and $\hat{c}$ is a conditional clause, then $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$.

6. If $\hat{a}_1 = (\hat{x}'' = b)$, $\hat{a}_1 \lll \hat{a}_0$, and $\hat{x}'' \neq \hat{x}$, then $\hat{D}(\hat{x}, \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$.

7a. If $\hat{a}_1 = (\hat{x}'' \overset{\hat{c}\downarrow}{=} \hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$, $\hat{x}'' \neq \hat{x}$, $\hat{c}$ is an application clause, and $\textsc{IsTop}(\hat{c}, \hat{C})$, then $\hat{D}(\hat{x}_f, \hat{a}_1, \hat{x} \,||\, \hat{X}, \textsc{Pop}(\hat{C})) \subseteq \hat{V}$.

7b. If $\hat{a}_1 = (\hat{x}'' \overset{\hat{c}\downarrow}{=} \hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$, $\hat{x}'' \neq \hat{x}$ and $\hat{c}$ is a conditional clause, then $\hat{D}(\hat{x}, \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$.
(where in the above $\textsc{RV}(\hat{e}) = \hat{x}$ if $\hat{e} = [\hat{c}, \ldots, \hat{x} = \hat{b}]$, i.e. $\hat{x}$ is the return variable of $\hat{e}$.)

Note this is a well-formed inductive definition by inspection. Each of the clauses above represents a different case in the reverse search for a variable; we now give clause-by-clause intuitions. **(1)** We finally arrived at a definition of the variable $\hat{x}$ and so it must be in the result set. **(2)** The variable $\hat{x}$ we are searching for has a function value, and unlike clause (1) there is a non-empty lookup stack. This means the variable on top of the lookup stack, $\hat{x}_1$, was a non-local and was pushed on to the non-local stack while searching for the definition of the function it resides in. That function definition, $\hat{x} = \hat{f}$, has now been found, and so we may continue to search for $\hat{x}_1$ from the current point in the graph. **(3)** We have found a definition of $\hat{x}$ but it is defined to be another variable $\hat{x}'$. We transitively switch to looking for $\hat{x}'$. **(4a)** We have reached the start of the function body and the variable $\hat{x}$ we are searching for was the formal argument $\hat{x}'$. So, continue by searching for $\hat{x}'$ from the call site. The IsTop clause constrains this stack frame exit to align with the frame we had last entered (in reverse). **(4b)** This is the case clause version of the previous. Case clauses can be viewed as inlined functions aligned by program context so use of $\hat{C}$ is not necessary for alignment. **(5a)** We have reached a return copy which is assigning our variable x, so to look for x we need to continue by looking for x' inside this function. Push $\hat{c}$ on the stack since we are now entering the body (in reverse) via that call site. For a more accurate analysis, the "provided" line additionally requires that we *only* "walk back" into function(s) that could have reached this call site; so, we launch a subordinate lookup of $\hat{x}_f$ and constrain $\hat{a}_1$ accordingly. **(6)** Here the previous clause is not a match so the search continues at any predecessor node. Note this will chain past function/match call sites which did not return the variable $\hat{x}$ we are looking for. This is sound in a pure functional language; when we address state in Section 5.3, we will enter such a function to verify an alias to our variable was not assigned. **(7a)** The precondition means we have reached the beginning of a function body and did not find a definition for the variable $\hat{x}$. In this case, we switch to searching for the clause that defined the function body we are exiting, which is $\hat{x}_f$, and push $\hat{x}$ on the non-locals stack. Once the defining point of $\hat{x}_f$ is found, $\hat{x}$ will be popped from the non-locals stack and we will resume searching for it. The IsTop clause constrains the stack frame being exited to align with the frame we had last entered (in reverse). **(7b)** This is the case clause variation of the previous; as with (4b) above, the stack is not needed for conditional alignment – there is syntactically only one entry and exit.

3.2 Abstract Evaluation

We are now ready to present the single-step abstract evaluation relation on dependency graphs. Like [28, 10] etc, it is a graph-based notion of evaluation, but where function bodies are never copied – a single body is shared.

3.2.1 Active nodes

In order to preserve standard evaluation order we define the notion of an active node, Active – only nodes with all previous nodes already executed can fire. This serves a purpose similar to an evaluation context in operational semantics [4].

$$\textsc{Application}\quad \frac{\hat{c} = (\hat{x}_1 \texttt{ = } \hat{x}_2\ \hat{x}_3) \qquad \textsc{Active}(\hat{c}, \hat{D}) \qquad \hat{f} \in \hat{D}(\hat{x}_2, \hat{c}) \qquad \hat{v} \in \hat{D}(\hat{x}_3, \hat{c})}{\hat{D} \xrightarrow{\hat{}}{}^{1}\ \hat{D} \cup \textsc{Wire}(\hat{c}, \hat{f}, \hat{x}_3, \hat{x}_1)}$$

$$\textsc{Record Conditional True}\quad \frac{\hat{c} = (\hat{x}_1 \texttt{ = } \hat{x}_2 \texttt{ ~ } \hat{r} \texttt{ ? } \hat{f}_1 \texttt{ : } \hat{f}_2) \qquad \textsc{Active}(\hat{c}, \hat{D}) \qquad \hat{r}' \in \hat{D}(\hat{x}_2, \hat{c}) \qquad \hat{r} \subseteq \hat{r}'}{\hat{D} \xrightarrow{\hat{}}{}^{1}\ \hat{D} \cup \textsc{Wire}(\hat{c}, \hat{f}_1, \hat{x}_2, \hat{x}_1)}$$

$$\textsc{Record Conditional False}\quad \frac{\hat{c} = (\hat{x}_1 \texttt{ = } \hat{x}_2 \texttt{ ~ } \hat{r} \texttt{ ? } \hat{f}_1 \texttt{ : } \hat{f}_2) \qquad \textsc{Active}(\hat{c}, \hat{D}) \qquad \hat{v} \in \hat{D}(\hat{x}_2, \hat{c}) \qquad \hat{v} \text{ of form } \hat{r}' \text{ only if } \hat{r} \not\subseteq \hat{r}'}{\hat{D} \xrightarrow{\hat{}}{}^{1}\ \hat{D} \cup \textsc{Wire}(\hat{c}, \hat{f}_2, \hat{x}_2, \hat{x}_1)}$$

Figure 11 Abstract Evaluation Rules.

▶ **Definition 3.5.** $\textsc{Active}(\hat{a}', \hat{D})$ iff path $\textsc{Start} \ll \hat{a}_1 \ll \ldots \ll \hat{a}_n \ll \hat{a}'$ appears in $\hat{D}$ such that no $\hat{a}_i$ is of one of the forms $\hat{x} \texttt{ = } \hat{x}'\ \hat{x}''$ or $\hat{x} \texttt{ = } \hat{x}' \texttt{ ~ } \hat{p} \texttt{ ? } \hat{f} \texttt{ : } \hat{f}'$. We write $\textsc{Active}(\hat{a}')$ when $\hat{D}$ is understood from context.

3.2.2 Wiring

Recall from Section 2 how function application required the concrete function body to be "wired" directly in to the call site node, and how additional nodes were added to copy in the argument and out the result. The following definition accomplishes this.

▶ **Definition 3.6.** Let $\textsc{Wire}(\hat{c}', \texttt{fun}\ \hat{x}_0 \texttt{ -> (} [\hat{c}_1, \ldots, \hat{c}_n] \texttt{)}, \hat{x}_1, \hat{x}_2) =$
$\textsc{Preds}(\hat{c}') \ll (\hat{x}_0 \overset{\hat{c}'\downarrow}{=} \hat{x}_1) \ll \hat{c}_1 \ll \ldots \ll \hat{c}_n \ll (\hat{x}_2 \overset{\hat{c}'\uparrow}{=} \textsc{RV}([\hat{c}_n])) \ll \textsc{Succs}(\hat{c}')$.

$\hat{c}'$ here is the call site, and $\hat{c}_1 \ll \ldots \ll \hat{c}_n$ is the wiring of the function body. The Preds/Succs functions (given above in Definition 3.1) reflect how we simply wire to the existing predecessor(s) and successor(s).

Next, we define the abstract small-step relation $\xrightarrow{\hat{}}{}^{1}$ on graphs, see Figure 11.

▶ **Definition 3.7.** We define the small step relation $\xrightarrow{\hat{}}{}^{1}$ to hold if a proof exists in the system in Figure 11. We write $\hat{D}_0 \xrightarrow{\hat{}}{}^{*} \hat{D}_n$ to denote $\hat{D}_0 \xrightarrow{\hat{}}{}^{1} \hat{D}_1 \xrightarrow{\hat{}}{}^{1} \ldots \xrightarrow{\hat{}}{}^{1} \hat{D}_n$.

The evaluation rules are straightforward after the above preliminaries. For application, if $\hat{c}$ is a call site that is an active redex, lookup of the function variable $\hat{x}_2$ returns function body $\hat{f}$ and *some* value $\hat{v}$ can be looked up at the argument position, we may wire in $\hat{f}$'s body to this call site. Note that $\hat{v}$ is not added to the graph, it is only observed here to constrain evaluation order to be call-by-value. The case clause rules are similar.

3.3 Decidability

We begin with the computability of the variable lookup operation, the source of all of the computational complexity of the analysis.

▶ **Lemma 3.8.** *For any context stack model Σ with a finite $\hat{\boldsymbol{C}}$ and computable* Push, Pop *and* IsTop *operations, $\hat{D}(\hat{x}_0, \hat{a})$ is a computable function.*

Proof. This proof proceeds by reduction to the problem of reachability in a push-down system (PDS) accepting by empty stack. A push-down system is a push-down automaton (PDA) with an empty input alphabet; PDA/PDS reachability is polynomial-time [3, 1].

We define a PDS in which each state is a pair between a program point and a context stack (of which there are finitely many); the initial state is the pair $\hat{a} \times \epsilon$. The stack of the PDS corresponds roughly to the lookup stack and the current variable of the lookup operation: each variable of the source program is a member of the stack alphabet and the initial PDS stack is then $[\hat{x}_0]$ for computing $\hat{D}(\hat{x}_0, \hat{a})$. Each clause in Definition 3.4 (*except* clause 5a, which is discussed below) directly corresponds to a collection of transitions in the PDS. For instance, suppose $\hat{a}_1 = (\hat{x}' = \hat{v}')$ and $\hat{a}_1 \ll \hat{a}_0$; then, by clauses 1 and 2, each node $\hat{a}_0 \times \hat{C}$ in the PDS transitions to $\hat{a}_1 \times \hat{C}$ by popping $\hat{x}'$ (and pushing no symbols).

Clause 5a of Definition 3.4 requires special handling due to the additional "provided" constraint which must analyze a subordinate invocation of the lookup function. To perform this subordinate lookup in a way that affects the PDS transitions, we include the program's values and the PDS's own states in the PDS stack grammar. A subordinate lookup is started by pushing the current state and the lookup variable onto the stack. When a value is found, it is pushed onto the stack. Finally, *every* PDS state can transition to *any* target PDS state by popping the target PDS state from the stack. With some encoded stack reordering, the subordinate lookup returns us to the state in which it started with the resulting value on the top of the stack.

Given the above, it suffices to show that the values given by $\hat{D}(\hat{x}_0, \hat{a})$ are exactly those of the nodes reachable with an empty stack in the PDS. Let $\hat{V} = \hat{D}(\hat{x}_0, \hat{a})$ and let $\hat{V}'$ be the values in the PDS's reachable nodes. For each $\hat{v} \in \hat{V}$, $\hat{v} \in \hat{V}'$ follows directly by induction on the size of the proof of $\hat{v} \in \hat{V}$, constructing a path through the PDS at each step. For each $\hat{v} \in \hat{V}'$, $\hat{v} \in \hat{V}$ follows directly by induction on the length of any path in the PDS which reaches the node containing $\hat{v}$. ◀

3.3.1 Some simple context stack models

A natural family of context stacks is one where stacks are the k latest frames; to also admit the unbounded case we let k range over $\mathbf{Nat} \cup \omega$ for ω the first limit ordinal. We let $[\hat{c}', \hat{c}_1, \ldots, \hat{c}_n]\lceil k$ denote $[\hat{c}', \hat{c}_1, \ldots, \hat{c}_m]$ for $m = \min(k, n)$.

▶ **Definition 3.9.** Fixing k, we define context stack model Σ_k as having stack set $\hat{\boldsymbol{C}}$ be the set of all lists of $\hat{c}$ occurring in the program up to length k, and with the stack operations as follows:

- $\textsc{Push}(\hat{c}', [\hat{c}_1, \ldots, \hat{c}_n]) = [\hat{c}', \hat{c}_1, \ldots, \hat{c}_n]\lceil k$
- $\textsc{Pop}([\hat{c}_1, \ldots, \hat{c}_n]) = [\hat{c}_2, \ldots, \hat{c}_n]$ if $n > 0$; $\textsc{Pop}([]) = []$.
- $\textsc{IsTop}(\hat{c}', [\hat{c}_1, \ldots, \hat{c}_n])$ is true if $\hat{c}' = \hat{c}_1$ or if $n = 0$; it is false otherwise.

The term "kDDPA" we use to refer to DDPA with the context stack model Σ_k.

▶ **Lemma 3.10.** *Fixing Σ to some Σ_k for fixed constant k, $\hat{D}(\hat{x}, \hat{a}_0)$ is computable in polynomial time in the number of nodes in graph $\hat{D}$.*

Proof. Let g be the number of nodes in graph $\hat{D}$. By inspection, the PDS built in the proof of Lemma 3.8 will have a number of states which is of order the product of g and the number of context stacks in Σ_k. For Σ_k, the number of stacks is of order $O(g^k)$. Since PDS reachability is computable polynomially in the number of nodes in the PDS, the result immediately follows. ◀

Note that if k was not fixed and was in fact increasing with the size of the program, it would become exponential.

▶ **Lemma 3.11.** *Variable lookup is monotonic; that is, for any $\hat{x}$ and $\hat{a}$, if $\hat{D}_1 \subseteq \hat{D}_2$ then $\hat{D}_1(\hat{x}, \hat{a}) \subseteq \hat{D}_2(\hat{x}, \hat{a})$.*

Proof. Variable lookup is encodable as a PDS reachability problem (see Lemma 3.8) and the PDS grows monotonically with the graph $\hat{D}$. PDS reachability grows monotonically with the PDS. Therefore, the set of results from variable lookup grows monotonically with the graph $\hat{D}$. ◀

▶ **Lemma 3.12.** *The evaluation relation $\xrightarrow{\hat{}}{}^*$ is confluent.*

Proof. By inspection of Figure 11, the single-step rules only add to graph $\hat{D}$. The Active relation is also clearly monotone: any enabled redex is never disabled. Confluence is trivial from these two facts. ◀

▶ **Lemma 3.13.** *The evaluation relation $\xrightarrow{\hat{}}{}^*$ is terminating,* i.e. *for any $\hat{D}_0$ there exists a $\hat{D}_n$ such that $\hat{D}_0 \xrightarrow{\hat{}}{}^* \hat{D}_n$ and if $\hat{D}_n \xrightarrow{\hat{}}{}^* \hat{D}_{n+1}$, $\hat{D}_n = \hat{D}_{n+1}$. Furthermore, n is polynomial in the size of the initial program.*

Proof. By inspection of Figure 11, we have for any step $\hat{D}' \xrightarrow{\hat{}}{}^1 \hat{D}''$ that $\hat{D}' \subseteq \hat{D}''$. The only new nodes that can be added to $\hat{D}$ in the course of evaluation are the entry/exit nodes $\hat{x}' \overset{\hat{c}\downarrow}{=} \hat{x} \;/\; \hat{x} \overset{\hat{c}\uparrow}{=} \hat{x}'$, and only one of each of those nodes can exist for each call site (or case clause) / function body pair in the source program: $\hat{c}$ is the call site, and $\hat{x}$ / $\hat{x}'$ are variables in that call site and function body source, respectively. So, the number of nodes that can be added is always less then two times the square of the size of the original program. A similar argument holds for added edges. ◀

We let $\hat{D} \downarrow \hat{D}'$ abbreviate $\hat{D} \xrightarrow{\hat{}}{}^* \hat{D}'$ such that $\hat{D}' \xrightarrow{\hat{}}{}^1 \hat{D}'$. We write $e \downarrow \hat{D}$ to abbreviate $\textsc{Embed}(e) \downarrow \hat{D}$; this means the analysis of e returns graph $\hat{D}$. Given the pieces assembled above, it is now easy to prove that the analysis is polynomial-time.

▶ **Theorem 3.14.** *Fixing Σ to be some Σ_k and fixing some expression e, the analysis result $\hat{D}$, where $e \downarrow \hat{D}$, is computable in time polynomial in the size of e.*

Proof. By Lemma 3.10, each lookup operation takes poly-time. The evaluation rules are trivial computations besides the required lookups and, by Lemma 3.13, there are polynomially many evaluation steps before termination. Thus $e \downarrow \hat{D}$ is computable in poly-time. ◀

4 Soundness

We now establish soundness of the analysis defined in the previous section. In forward program analyses the alignment between the operational semantics and analysis is fairly close and so soundness is not particularly difficult, but here there is a larger gap. We cross this river by throwing a stone in the middle: along with defining a mostly-standard operational semantics we build a *graph-based operational semantics* which creates a *concrete* call graph of the program run that is more directly aligned with the analysis. Soundness is then shown by proving the standard and graph-based operational semantics equivalent and by showing the analysis sound with respect to the graph-based operational semantics.

In this section we first present the standard operational semantics, then the graph-based operational semantics and its equivalence to the standard one, and finally we prove soundness.

$$\textsc{Lookup}\quad \frac{x_2 = v \in E}{E\,||[x_1 = x_2]\,||\,e \longrightarrow^1 E\,||[x_1 = v]\,||\,e} \qquad \textsc{Application}\quad \frac{x_2 = f \in E \qquad x_3 = v \in E \qquad f' = \textsc{fr}(x_1, f)}{E\,||[x_1 = x_2\ x_3]\,||\,e \longrightarrow^1 E\,||\,\textsc{Wire}(f', v, x_1)\,||\,e}$$

$$\textsc{Record Conditional True}\quad \frac{x_2 = r' \in E \qquad r \subseteq r' \qquad f_1' = \textsc{fr}(x_1, f_1)}{E\,||[x_1 = x_2 \sim r\ ?\ f_1 : f_2]\,||\,e \longrightarrow^1 E\,||\,\textsc{Wire}(f_1', r', x_1)\,||\,e}$$

$$\textsc{Record Conditional False}\quad \frac{x_2 = v \in E \qquad v \text{ of the form } r' \text{ only if } r \not\subseteq r' \qquad f_2' = \textsc{fr}(x_1, f_2)}{E\,||[x_1 = x_2 \sim r\ ?\ f_1 : f_2]\,||\,e \longrightarrow^1 E\,||\,\textsc{Wire}(f_2', r', x_1)\,||\,e}$$

Figure 12 Small-Step Evaluation.

4.1 Standard Operational Semantics

The operational semantics appears in Figure 12, as a small step relation $e \longrightarrow^1 e'$. In many ways the operational semantics is standard, but due to our use of an A-normal form it is neither precisely substitution-based nor environment-based. It is more substitution-based in spirit since function bodies are inlined. Although variable lookup is via an environment, all names in that environment are deterministically freshened and so no variable shadowing ever arises; so, although variable lookup might appear to be dynamically scoped, the absence of shadowing ensures static scoping. This model of evaluation is designed to integrate well with the graph-based semantics of the next sub-section.

We must freshen variables as they are introduced to the expression to preserve the invariant that each variable is uniquely defined and no shadowing occurs. We give a somewhat nonstandard definition of freshening which is deterministic, so as to make alignments easier. We take $\textsc{fr}(x', x)$ to yield another variable x''; we require that $\textsc{fr}(-,-)$ is injective and that its codomain does not include variables appearing in the initial program. Here, x is the variable to be freshened and x' is the point in the program at which it is freshened. For informal illustration, one concrete freshening function could be $\textsc{fr}(x', x) = x^{x'}$. We overload $\textsc{fr}(x', v)$ to indicate the freshening of all variables bound in v. The rules for the small step relation are given in Figure 12. Here, *wiring* is the process of inlining a function body; for this we use the following auxiliary function.

▶ **Definition 4.1.** Let $\textsc{Wire}(\texttt{fun}\,x\,\texttt{->}\,\texttt{(}\,e\,\texttt{)}, v, x') = [x = v]\,||\,e\,||[x' = \textsc{rv}(e)]$.

▶ **Definition 4.2.** We define the small step relation $\longrightarrow^1$ to hold if a proof exists in the system in Figure 12. We write $e_0 \longrightarrow^* e_n$ to denote $e_0 \longrightarrow^1 e_1 \longrightarrow^1 \ldots \longrightarrow^1 e_n$.

▶ **Definition 4.3.** If, for some expression e not of the form E, there exists no e' such that $e \longrightarrow^1 e'$, then we say that e is *stuck*. For any e'' such that $e'' \longrightarrow^* e$, we say that e'' *becomes stuck*.

4.2 Graph-Based Operational Semantics

We now define the graph-based operational semantics, and prove it to be equivalent to the standard operational semantics just defined.

$$
\begin{array}{rcll}
V & ::= & \{v, \ldots\} & \textit{value sets} \\
a & ::= & c \mid x \overset{\mathrm{c}\downarrow}{=} x \mid x \overset{c\uparrow}{=} x \mid & \textit{annotated clauses} \\
 & & \textsc{Start} \mid \textsc{End} & \\
d & ::= & a \ll a & \textit{dependencies}
\end{array}
\qquad
\begin{array}{rcll}
D & ::= & \{d, \ldots\} & \textit{dependency graphs} \\
X & ::= & [x, \ldots] & \textit{lookup stacks} \\
C & ::= & [c, \ldots] & \textit{context stacks}
\end{array}
$$

Figure 13 Graph-Based Evaluation Grammar.

The graph-based operational semantics is structurally very similar to the analysis. The primary difference is function bodies here are *copied* to make new graph structure (as opposed to the analysis, in which only one copy of each function body exists in the graph). Otherwise, the graph-based operational semantics and the analysis are nearly identical, including the use of context stacks, lookup operations, and so on. Since many of these definitions are so similar, we will be much more brief here; we assume that the reader has absorbed the analysis definitions.

The graph-based operational semantics differs from the standard operational semantics of Section 4.1 in the following dimensions:

1. the total ordering of the list becomes a partial ordering here;
2. alias clauses $x = x'$ are resolved lazily rather than eagerly, and
3. the graph is monotonically increasing; that is, in each place that the standard semantics *replaces* a call site, the graph-based semantics builds a path *around* the call site.

The new grammar constructs needed for the graph-based operational semantics appear in Figure 13.

We write $\textsc{Embed}(e)$ to denote the initial embedding of an expression e into a dependency graph D. This definition is identical to analysis Definition 3.2 with hats removed from the metavariables. Similarly, we also will use the notational sugar of Notation 3.1 in perfect analogy.

4.2.1 Variable value lookup

The definition of lookup proceeds with respect to the context stack C which is directly aligned with the corresponding stack $\hat{C}$ of the analysis. This stack is not necessary in the operational semantics (as copying of function bodies removes all of the ambiguity that was in the analysis); we retain it for alignment. Unlike the analysis, we need not bound C; we therefore fix the context stack model of the operational semantics to the equivalent of the unbounded model Σ_ω in this grammar.

Lookup finds the value of a variable starting from a given graph node; this definition is a near-exact parallel of Definition 3.4 and the reader is referred there for intuitions. We let $D(x, a_0)$ abbreviate $D(x, a_0, [], [])$.

▶ **Definition 4.4.** Given graph D, $D(x, a_0, X, C)$ is the function returning the least set of values V satisfying the following conditions:

1. If $a_1 = (x = v)$, $a_1 \lll a_0$, and $X = []$, then $v \in V$.
2. If $a_1 = (x = f)$, $a_1 \lll a_0$, and $X = [x_1, \ldots, x_n]$ for $n > 0$, then $D(x_1, a_1, [x_2, \ldots, x_n], C) \subseteq V$.
3. If $a_1 = (x = x')$ and $a_1 \lll a_0$, then $D(x', a_1, X, C) \subseteq V$.

4a. If $a_1 = (x \overset{\mathrm{c}\downarrow}{=} x')$, $a_1 \lll a_0$, c is an application clause, and $\textsc{IsTop}(c, C)$, then $D(x', a_1, X, \textsc{Pop}(C)) \subseteq V$.

4b. If $a_1 = (x \overset{\mathrm{c}\downarrow}{=} x')$, $a_1 \lll a_0$ and c is a conditional clause, then $D(x', a_1, X, C) \subseteq V$.

$$\textsc{Application}\quad \frac{c = (x_1 \texttt{ = } x_2\ x_3) \qquad \textsc{Active}(c, D) \qquad f \in D(x_2, c) \qquad v \in D(x_3, c) \qquad f' = \textsc{fr}(x_1, f)}{D \longrightarrow^1 D \cup \textsc{Wire}(c, f', x_3, x_1)}$$

$$\textsc{Record Conditional True}\quad \frac{c = (x_1 \texttt{ = } x_2 \texttt{ ~ } r \texttt{ ? } f_1 \texttt{ : } f_2) \qquad \textsc{Active}(c, D) \qquad r' \in D(x_2, c) \qquad r \subseteq r' \qquad f_1' = \textsc{fr}(x_1, f_1)}{D \longrightarrow^1 D \cup \textsc{Wire}(c, f_1', x_2, x_1)}$$

$$\textsc{Record Conditional False}\quad \frac{\begin{array}{c} c = (x_1 \texttt{ = } x_2 \texttt{ ~ } r \texttt{ ? } f_1 \texttt{ : } f_2) \\ \textsc{Active}(c, D) \qquad v \in D(x_2, c) \qquad v \text{ of form } r' \text{ only if } r \not\subseteq r' \qquad f_2' = \textsc{fr}(x_1, f_2) \end{array}}{D \longrightarrow^1 D \cup \textsc{Wire}(c, f_2', x_2, x_1)}$$

Figure 14 Graph Evaluation Rules.

5a. If $a_1 = (x \overset{c\uparrow}{=} x')$, $a_1 \lll a_0$ and $c = (x_r \texttt{ = } x_f\ x_v)$, then $D(x', a_1, X, \textsc{Push}(c, C)) \subseteq V$, provided `fun` x'' `->` `(` e `)` $\in D(x_f, c, [], C)$ and $x' = \textsc{rv}(e)$.

5b. If $a_1 = (x \overset{c\uparrow}{=} x')$, $a_1 \lll a_0$ and c is a conditional clause, then $D(x', a_1, X, C) \subseteq V$.

6. If $a_1 = (x'' \texttt{ = } b)$, $a_1 \lll a_0$, and $x'' \neq x$, then $D(x, a_1, X, C) \subseteq V$.

7a. If $a_1 = (x'' \overset{c\downarrow}{=} x')$, $a_1 \lll a_0$, $x'' \neq x$, c is an application clause, and $\textsc{IsTop}(c, C)$, then $D(x_f, a_1, x \,||\, X, \textsc{Pop}(C)) \subseteq V$.

7b. If $a_1 = (x'' \overset{c\downarrow}{=} x')$, $a_1 \lll a_0$, $x'' \neq x$ and c is a conditional clause, then $D(x, a_1, X, C) \subseteq V$.

where $\textsc{rv}(e) = x$ if $e = [c, \ldots, x \texttt{ = } b]$, i.e. x is the return variable of e.

4.2.2 The evaluation relation

The definition of an active node, Active, exactly parallels the analysis version of Definition 3.5, simply remove the hats from the metavariables. We write $\textsc{Active}(a')$ when D is understood from context. Wiring is also defined in perfect analogy with Definition 3.6 so is omitted here. The small-step relation $\longrightarrow^1$ on graphs is defined in Figure 14. (Note we overload symbol $\longrightarrow^1$ for the list and graph operational semantics, it is clear from context which relation is intended.)

▶ **Definition 4.5.** We define the small step relation $\longrightarrow^1$ to hold if a proof exists in the system in Figure 14. We write $D_0 \longrightarrow^* D_n$ to denote $D_0 \longrightarrow^1 D_1 \longrightarrow^1 \ldots \longrightarrow^1 D_n$.

We also define a notion of "stuckness" for graphs in parallel with the standard operational semantics.

These rules are very similar to the analysis transition rules in Section 3.2; we refer the reader to the descriptions there. Here we comment on a few points unique to the operational semantics not found in the analysis.

We are precise about freshening in these rules. Because the $\textsc{fr}(x, -)$ function is deterministically freshening based on the call site x, it will always generate the same result for the same call site and value. This means that e.g. each use of the Application rule is idempotent and the graph-based operational semantics is deterministic.

Observe how Figure 14 is defining a "reduction" relation which is in fact monotonically *increasing*, an unusual property compared to standard operational semantics.

4.3 Proving Equivalence of Operational Semantics

The overall proof of soundness relies upon showing that these two systems of operational semantics are equivalent. To demonstrate this, we must show several properties of the graph-based operational semantics which are less obvious than in the standard operational semantics due to the structure of the graph. As stated above, the real differences between these systems are (1) the partial ordering of clauses, (2) lazy resolution of alias clauses, and (3) that the graph grows monotonically instead of replacing applications and conditionals. Nonetheless, the graph representation "runs" in the same way that expressions do: there is a unique clause which is evaluated next, it may cause the introduction of more clauses, and *complex clauses* (applications and conditionals) are handled by wiring and evaluating the appropriate function body.

We use this reasoning to establish a bisimulation $\cong$ between an expression and its embedding and then show that this bisimulation is preserved as evaluation proceeds. The bisimulation and corresponding preservation proof are tedious and intuitive, so we exclude them for reasons of space. The key equivalence lemma is stated as follows.

▶ **Lemma 4.6** (Equivalence of standard and graph-based semantics)**.** *If $e \cong D$, then*

1. *If $e \longrightarrow^1 e'$ then $D \longrightarrow^* D'$ such that $e' \cong D'$.*
2. *If $D \longrightarrow^1 D'$ then $e \longrightarrow^* e'$ such that $e' \cong D'$.*

This equivalence is one-to-many to accommodate the differences between the two operational semantics relations. The graph opsems lazily resolves aliases, meaning that several steps of the standard opsems may be required to propagate a value that the graph would find via lookup (Definition 4.4). Likewise, a single lookup step of the standard opsems may require no changes to the graph to maintain bisimulation.

4.4 Soundness of the Analysis

We now show that the analysis simulates the graph-based operational semantics, $D \preccurlyeq \hat{D}$. This proof is easy, as the operational semantics graph can be projected on to an analysis graph by inverting the variable freshening function. We formally define this inversion as follows:

▶ **Definition 4.7.** The *origin* of a variable x is x itself if x is not in the codomain of $\textsc{fr}(-,-)$. Otherwise, let $\textsc{fr}(x'', x') = x$ (for unique x'' and x', as $\textsc{fr}(-,-)$ is injective). Then the origin of x is the origin of x'.

We next define the simulation between evaluation graphs and analysis graphs. The graph operational semantics was designed explicitly to make this simulation direct.

▶ **Definition 4.8** (Simulation relation)**.** Let f be the natural graph-and-clause homomorphism from an evaluation graph D to an analysis graph $\hat{D}$ which maps variables to their origins. We say that D is simulated by $\hat{D}$ (written $D \preccurlyeq \hat{D}$) iff $f(D) = \hat{D}$.

▶ **Lemma 4.9** (Soundness)**.** *If $D \preccurlyeq \hat{D}$ and $D \longrightarrow^1 D'$, then $\hat{D} \mathrel{\hat{\longrightarrow}}^1 \hat{D}'$ with $D' \preccurlyeq \hat{D}'$.*

Proof. By case analysis on the rule used to prove $D \longrightarrow^1 D'$. In particular, the premises of a corresponding rule $\hat{D} \mathrel{\hat{\longrightarrow}}^1 \hat{D}'$ can be proven using the premises of $D \longrightarrow^1 D'$ and the simulation. ◀

5 Extensions

In this section, we outline three extensions: full records, path sensitivity, and mutable state. Our goal here is to show there is no fundamental limitation to the model given in the previous sections: DDPA can in principle be extended to the full feature set of a realistic programming language. Here for simplicity of presentation we take each feature one at a time; in a language where they are all simultaneously added there are additional feature interaction cases that must be addressed.

5.1 Records

The "records" presented thus far have no projection operation since there are no values to project; here we outline an extension to records with projection. Consider a lookup of variable $\hat{x}$: if $\hat{x}$ is defined as $\hat{x} = \hat{x}'.\ell$, then this necessitates a record projection; $\hat{x}'$ itself may also necessitate further projection, and so on. In the general case, there could be a continuation stack of record projections to be performed. This is similar to the non-local lookup stack of our analysis, and not coincidentally: non-locals may be encoded in terms of records via closure conversion.

In light of this connection, we can extend lookup to support full records using a single stack $\hat{X}$ (now more appropriately thought of as a *continuation stack*) to handle both data destructors as well as non-local variables. We define the grammar of lookup actions to include variables $\hat{x}$ and projections of the form $.\ell$; we use $\hat{k}$ to range over these lookup actions and extend $\hat{X}$ to all lists of $\hat{k}$. We then augment Definition 3.4 with the following new clauses:

▶ **Definition 5.1.** We extend Definition 3.4 to full records by adding the following clauses.

9. If $\hat{a}_1 = (\hat{x} = \{\ell_1 = \hat{x}', \ldots\})$, $\hat{a}_1 \lll \hat{a}_0$, $\hat{X} = [.\ell_1, \hat{k}_2, \ldots, \hat{k}_n]$, then $\hat{D}(\hat{x}', \hat{a}_1, [\hat{k}_2, \ldots, \hat{k}_n], \hat{C}) \subseteq \hat{V}$.
10. If $\hat{a}_1 = (\hat{x} = \hat{x}'.\ell)$, $\hat{a}_1 \lll \hat{a}_0$, and $\hat{X} = [\hat{k}_1, \ldots, \hat{k}_n]$, then $\hat{D}(\hat{x}', \hat{a}_1, [.\ell, \hat{k}_1, \ldots, \hat{k}_n], \hat{C}) \subseteq \hat{V}$.

We also extend each of the clauses of Definition 3.4 to address lookup actions generally (rather than requiring lookup stacks to consist only of variables). Clause 10 above introduces the projection action $.\ell$ when we discover that we will need to project from the variable we find; clause 9 eliminates this projection action when the corresponding record value is found. Note that record pattern matching would also be desirable syntax, and would be possible by extending clause 8 of the original lookup definition; for brevity, we skip that here.

5.2 Filtering for path sensitivity

An additional level of expressiveness we desire from records is the ability to filter out paths that cannot have been taken. Consider, for instance, the recursive example in Figure 8. This example appears to be unsafe based on the analysis result: on line 7, we project `l` from the function's argument. The condition on line 4 makes this safe – only records with `l` labels may reach line 7 – but the analysis is unaware of this and so erroneously believes that the record `{}` may reach that point.

We address this incompleteness using *filters*: sets of patterns which the value must match in order to be considered relevant. In the above example, for instance, looking up `v` from line 7 leads us to look up `a` from line 4. Because our path moved backwards through the first clause of a conditional, however, we know that we may ignore any values we discover

which do not match the `{1}` pattern. This is key to enabling path sensitivity and preventing spurious errors; with filters, DDPA correctly identifies the recursion example as safe.

We can formalize path sensitivity in DDPA by keeping track of sets of accumulated patterns in our lookup function, and disallowing matches not respecting the patterns they passed through. We use Π^+ and Π^- to range over sets of patterns which a discovered value *must* or *must not* match, respectively. Formally, path-sensitive DDPA is possible by modifying the lookup function as follows:

▶ **Definition 5.2.** We modify Definition 3.4 by adding to the lookup function two parameters of the forms Π^+ and Π^-. Each existing clause of that definition passes these arguments unchanged. We additionally replace clauses 1 and 4b of Definition 3.4 with:

1. If $\hat{a}_1 = (\hat{x} \texttt{=} \hat{v})$, $\hat{a}_1 \lll \hat{a}_0$, $\hat{X} = []$, and $\hat{v}$ matches all patterns in Π^+ and no patterns in Π^-, then $\hat{v} \in \hat{V}$.

4b1. If $\hat{a}_1 = (\hat{x}' \overset{\hat{c}\downarrow}{\texttt{=}} \hat{x}_1)$, $\hat{a}_1 \lll \hat{a}_0$, $\hat{c} = (\hat{x}_2 \texttt{=} \hat{x}_1 \texttt{\~{}} \hat{p} \texttt{?} \hat{f}_1 \texttt{:} \hat{f}_2)$, $\hat{f}_1 = \texttt{fun}\ \hat{x}' \texttt{->} (\hat{e})$, and $\hat{x} \in \{\hat{x}_1, \hat{x}'\}$, then $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \hat{C}, \Pi^+ \cup \{\hat{p}\}, \Pi^-) \subseteq \hat{V}$.

4b2. If $\hat{a}_1 = (\hat{x}' \overset{\hat{c}\downarrow}{\texttt{=}} \hat{x}_1)$, $\hat{a}_1 \lll \hat{a}_0$, $\hat{c} = (\hat{x}_2 \texttt{=} \hat{x}_1 \texttt{\~{}} \hat{p} \texttt{?} \hat{f}_1 \texttt{:} \hat{f}_2)$, $\hat{f}_2 = \texttt{fun}\ \hat{x}' \texttt{->} (\hat{e})$, and $\hat{x} \in \{\hat{x}_1, \hat{x}'\}$, then $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \hat{C}, \Pi^+, \Pi^- \cup \{\hat{p}\}) \subseteq \hat{V}$.

4b3. If $\hat{a}_1 = (\hat{x}' \overset{\hat{c}\downarrow}{\texttt{=}} \hat{x}_1)$, $\hat{a}_1 \lll \hat{a}_0$, $\hat{c} = (\hat{x}_2 \texttt{=} \hat{x}_1 \texttt{\~{}} \hat{p} \texttt{?} \hat{f}_1 \texttt{:} \hat{f}_2)$, and $\hat{x} \notin \{\hat{x}_1, \hat{x}'\}$, then $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \hat{C}, \Pi^+, \Pi^-) \subseteq \hat{V}$.

We then redefine $\hat{D}(\hat{x}, \hat{a})$ to mean $\hat{D}(\hat{x}, \hat{a}, [], [], \emptyset, \emptyset)$.

Revised clause 1 shows how the filters are used: any value not matching the positive filter is filtered out, and oppositely for the negative filter. The original clause 4 was the case where we reached the start of a function and search variable $\hat{x}$ was passed as the parameter; in that case, the clause continued by searching for the argument at the call site. The original clause 4b was the case where we reached the start of a case clause and search variable $\hat{x}$ was passed as the parameter; in that case, the clause continued by searching for the argument at the call site. Here, we have separated that rule into two cases. In 4b1, the function was the first branch of a conditional, so we know that any discovered value is only relevant if it matches the conditional's pattern. Thus, we add the pattern to the filter set to constrain it so. Clause 4b2 is the opposite case.

5.3 State

Lookup in the presence of state may also be performed using only a call graph. We consider here a variation of the presentation language which includes OCaml-style references with `ref` x / `!` x / x `<-` x syntax. There are several subtle issues that must be addressed. First, a simple linear search may not always give the correct answer; a cell containing a cell could have the inner cell mutated after the outer cell, which is out of order compared to the control flow sequence. So, we must define a branching search for references. Second, aliases may arise in the form of different variable names referring to the same cell. Lookup must not overlook aliases; otherwise, the lookup will be inaccurate and incomplete. So, explicit alias testing is needed to verify proper values are not being passed by.

Not only may DDPA be adapted to address state, but the alias analysis itself is not difficult using the same lookup routine. The following definition revises the lookup operation for state.

▶ **Definition 5.3.** Definition 3.4 is extended to a stateful language by adding the following clauses.

9. If $\hat{a}_1 = (\hat{x}$ `= !` $\hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$, then letting $\hat{V}' = \hat{D}(\hat{x}', \hat{a}_1, [], \hat{C})$, for each `ref` $\hat{x}'' \in \hat{V}'$, $\hat{D}(\hat{x}'', \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$.

10a. If $\hat{a}_1 = (\hat{x}'_1$ `=` $\hat{x}'_2$ `<-` $\hat{x}'_3)$, $\hat{a}_1 \lll \hat{a}_0$, and MayAlias$(\hat{x}, \hat{x}'_2, \hat{a}_1, \hat{C})$, then `ref` $\hat{x}'_3 \in \hat{V}$.

10b. If $\hat{a}_1 = (\hat{x}'_1$ `=` $\hat{x}'_2$ `<-` $\hat{x}'_3)$, $\hat{a}_1 \lll \hat{a}_0$, and MayNotAlias$(\hat{x}, \hat{x}'_2, \hat{a}_1, \hat{C})$, then $\hat{D}(\hat{x}, \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$.

In these clauses, the terms MayAlias and MayNotAlias refer to the following predicates:

- MayAlias$(\hat{x}_1, \hat{x}_2, \hat{a}, \hat{C})$ holds iff $\hat{V}' = \hat{D}(\hat{x}_1, \hat{a}, [], \hat{C})$, $\hat{V}'' = \hat{D}(\hat{x}_2, \hat{a}, [], \hat{C})$, and $\exists$`ref` $\hat{x}'' \in (\hat{V}' \cap \hat{V}'')$
- MayNotAlias$(\hat{x}_1, \hat{x}_2, \hat{a}, \hat{C})$ holds iff $\hat{V}' = \hat{D}(\hat{x}_1, \hat{a}, [], \hat{C})$, $\hat{V}'' = \hat{D}(\hat{x}_2, \hat{a}, [], \hat{C})$, and $\hat{V}' \neq \hat{V}''$ *or* $\hat{V}' \neq \{$`ref` $\hat{x}'\}$ *or* $\hat{V}'' \neq \{$`ref` $\hat{x}''\}$

Clause 9 handles dereferencing. It finds the `ref` values which may be in $\hat{x}'$ at this point in the program; it then returns to this point to find all of the values that those variables may contain. This new task is necessary since we want the value at the point the `!` happened.

Clauses 10a and 10b address cell updates. Clause 10a determines if the updated cell in $\hat{x}'_2$ *may* alias the cell we are looking up; if this is the case, the value assigned by the cell update may be our answer. Clause 10b addresses the case in which the updated cell *may* be different from the target of our lookup. This happens when the lookups of each variable yield different results or when they result in multiple cells – even if the sets of cells are equal, the orders in which the program modifies the cells might differ, so we take the conservative approach and call them different. MayAlias and MayNotAlias can be simultaneously satisfiable; when that happens, the analysis explores both clauses 10a and 10b.

Along with the above modifications, the existing clauses 5 and 6 need to be extended to support state. As written, clause 6 allows us to skip by call sites and pattern matches whose output do not match the variable for which we are searching. This is sound in a pure system, but in the presence of side-effects we must explore these clauses to ensure that they did not affect the cell we are attempting to dereference. We thus modify clause 6 by prohibiting $\hat{b}$ from being a call site or pattern match. We require a new clause similar to clause 5 but for the case in which the search variable does not match the output variable. In that case, we proceed into the body of the function but in a "side-effect only" mode: we skip by every clause which is not a cell assignment or does not lead to one. We leave side-effect only mode once we leave the beginning of the function which initiated it.

6 Implementation

We have developed a proof-of-concept implementation of DDPA [17]; a copy of the implementation is archived as an artifact with this paper. This implementation analyzes the presentation language given in Section 2 extended to the proper record semantics described in Section 5. The implementation correctly generates all control-flow graphs and value lookup results given in the overview.

Although the proof-of-concept implementation demonstrates the behavior of DDPA, it is far from efficient and we are presently developing a performant implementation. Our implementation follows the lookup algorithm outlined in the proof of Lemma 3.8: we construct a push-down automaton modeling a non-deterministic backwards walk of the DDPA graph and then analyze this PDA for states reachable with an empty stack. There is a wealth of literature on PDA reachability algorithms [3, 1] which we can utilize to gain efficiency; in particular, [3] includes an algorithm for eliminating spurious nodes and edges which we have already integrated into the proof-of-concept implementation due to its simplicity.

7 Related Work

7.1 CFL-reachability

The core idea of our analysis can be viewed as extending first-order demand-driven CFL-reachability analyses [6] to the higher-order case: the analysis is centered around using a CFG, calls and returns are aligned, and lookup is computed lazily. Two issues make the higher-order analysis challenging: the CFG needs to be computed on-the-fly due to the presence of higher-order functions, and non-local variable lookup is subtle. The aforecited demand-driven analyses delve further into the trade-off between active propagation and demand-driven lookup, and this is something we plan to explore in future work. There are many other first-order program analyses with a demand-driven component; several use Datalog-style specification formats [19] including [29, 23].

Higher-order program analyses have been built which also incorporate call-return alignment [26, 8], but these higher-order analyses are not demand-driven and have a different algorithmic basis. We now contrast DDPA with these and other higher-order program analyses.

7.2 Higher-order program analysis

As mentioned in the Introduction, higher-order program analyses today are generally constructed by abstract interpretation [2], a finitization process on the operational semantics. We refer to these analyses as *forward* analyses to contrast with the backward-looking, demand-driven approach of DDPA.

In the most basic forward analysis, a new state is created for each new program point/store/environment, and the number of states grows too rapidly to be practical. Modifications including store widening, abstract garbage collection, polyvariance, and call-return alignment are thus added as means to obtain a better trade-off of expressiveness vs size of state space. DDPA in some sense comes from the opposite direction: the global state information is a graph isomorphic to the CFG which is polynomial in size and will be smaller than the state space of any forward analysis. However, variable lookup is of high complexity and we need to "spend" more space to make it more efficient.

We will expand by considering the different dimensions of analysis expressiveness in turn. We begin with flow-sensitivity, meaning the analysis is sensitive to the *order* of assignment operations. Forward analyses are initially flow-sensitive since each abstract state has its own store, but the standard store widening optimization unifies these stores and eliminates flow-sensitivity. One alternative is abstract garbage collection [14, 8], which requires a per-node store (since, if the store were global, there would be no garbage). Abstract garbage preserves flow-sensitivity while reducing abstract states caused by local data, but state explosions still occur with persistent data. DDPA is flow-sensitive and garbage-free by construction: it is flow-sensitive because variables are looked up with respect to a program point, and it is garbage-free vacuously (as no stores are ever created). In a practical implementation of DDPA, the caching of values will bring it closer to an abstract garbage collection approach, but abstract garbage collection is bottom-up as opposed to top-down: rather than deleting items proved unneeded, items are added only if they may be needed in the future.

Another important dimension of expressiveness is polyvariance: whether functions can take on different forms in different contexts of use. The classic higher-order polyvariance model is kCFA [22], which copies contours in analogy to forall-elimination in a polymorphic type system. But there are many routes to behavior that appears as polyvariance. Without store widening, flow-sensitivity alone can distinguish calling context and provide some

polyvariance. Additionally, call-return alignment can provide different contexts for different function calls. The example we gave in Figure 3, for instance, needs only call-return alignment to give polyvariant behavior in DDPA.

Call-return alignment was first explored in a higher-order context in subtype constraint theories [18]; this work also uses the demand-driven nature of CFL-reachability to optimize lookup. However, it is flow-insensitive, uses let-polymorphism only, and does not align non-local variables, and so is not in the same category of analysis. In the context of abstract interpretation, the first such algorithm was CFA2 [26], which was subsequently extended and refined in kPDCFA [8]. We compare primarily with the more recent kPDCFA here.

An important observation of [8] regarding kPDCFA was that, in the presence of abstract garbage collection, it is not possible to have an arbitrary stack for aligning calls and returns. That paper proposes a regular expression modeling of call stacks to make the analysis computable. We have a similar problem: our analysis finitizes one of its two stacks to conform to a PDA encoding. We define the family of analyses kDDPA (Definition 3.3) where k is the maximum stack depth for call-return alignments. Our finitization here is simpler than the regular expression approach of kPDCFA; the regular expression approach may work in our context and is a topic for future investigation.

DDPA achieves greater polyvariance solely from call-return alignment when compared with kPDCFA or CFA2. In kPDCFA, stack alignment can be used to provide a different context for function parameters, but non-local variables get no such advantage since they are not in the local stack context. In DDPA, the stack context is still applicable to non-local variable lookup as shown in Figure 5. In fact, we conjecture that, for a program with a maximal lexical nesting depth of c, the analysis $(k+c)$DDPA will be at least as expressive as kCFA (and should be more expressive due to alignment of calls and returns). The additional c levels are needed because, in the worst case, each extra lexical level requires the lookup of a non-local variable and thus a position on the lookup stack. This way, DDPA achieves with only stack alignment what forward analyses need both stack alignment and polyvariant contours to accomplish.

The run-time complexity of kDDPA can also be framed in terms of the expressiveness of non-local variable polyvariance. It is shown in [15] how non-locals are the (only) source of exponential behavior in kCFA [24]; in particular, if lexical nesting were assumed to be of some constant depth not tied to the size of the program, kCFA would not be exponential. The complexity of kDDPA comes from the other direction: for any fixed k, the algorithm kDDPA is polynomial; but k needs to be increased by one for each level of stack alignment we wish to achieve in non-local lookup. For pathologically nested programs, k must be on the order of the size of the program for kDDPA to avoid imprecision due to non-local lookup. Because kDDPA's complexity is exponential in k, such a non-local-precise kDDPA would be exponentially complex in that pathological case.

Related to this are provably polynomial context-sensitive analyses which, like kDDPA, restrict context-sensitivity in the case of high degree of lexical nesting [7, 15]. mCFA [15] is a polyvariant analysis hierarchy for functional languages that is provably polynomial in complexity. This is achieved by an analysis that "in spirit" is working over closure-converted source programs: by factoring out all non-local variable references, the worst-case behavior has also been removed. Unfortunately, this also affects the precision of the analysis: non-locals that are distinguished in kCFA are merged in mCFA. In kDDPA, the level of non-local precision is built into the constant k of how deep the run-time stack approximation is, so more precision is achieved as k increases. mCFA does not have this property: non-locals will always be monomorphised for any m.

Our current implementation is a proof-of-concept only; we plan to investigate ideas in [11, 8, 9] and other papers for more performant PDA reachability algorithms. Overall the trade-offs in performance and expressiveness are subtle and implementations will be necessary to decide how practically useful DDPA is.

8 Conclusions

In this paper, we have developed a demand-driven program analysis (DDPA) for higher-order programs which is centered around production of a call graph. DDPA needs *only* the call graph to look up variable values; the specification does not maintain any other structures. DDPA can be viewed as the adaptation of previous CFL-reachability-based demand-driven program analyses to higher-order programs. This adaptation required two key changes. First, it was necessary to incrementally construct the control-flow graph as the analysis proceeded; second, the lookup of non-local variables required special handling.

We believe DDPA shows promise primarily because it represents a significantly different approach compared with other higher-order analyses. A high-level analogy can be made with eager and lazy programming languages: it is a fundamental decision in language design which approach to take and there are significant trade-offs. We believe the demand-driven side of higher-order program analyses deserves further exploration.

We have established a polynomial-time bound on a higher-order program analysis which is both flow- and context-sensitive. The reduced global state size holds out promise for program verification tools: the fewer the states, the less overwhelming the workload will be for a model checker, theorem prover, or other verification strategy.

We present preliminary results here from a simple implementation to serve as a confirmation of correctness, but our implementation needs tuning and would benefit from a head-to-head comparison with some state-of-the-art analyses. Although DDPA's value lookup is novel, it shares the task of PDA reachability with the existing program analysis literature and so a rich body of work is available to be utilized in implementing DDPA efficiently.

We believe the analysis should scale well to other language features, and have presented outlines for extensions of the basic analysis to deep data structures, path-sensitivity, and state; we leave efficient decision procedures for the latter two to future work. We also intend to explore the development of extensions for other language features: exceptions and other control operators, concurrency, and modularity to name a few.

Acknowledgements. We thank Alex Rozenshteyn for motivating our interest in AAM [25] and related papers which then led to our formulation of DDPA. We thank David Van Horn for discussions helping us better understand the AAM work. We also thank Leandro Facchinetti for his detailed review of this paper and his contributions to the accompanying artifact.

References

1 Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR'97*, 1997.

2 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

3 Christopher Earl, Matthew Might, and David Van Horn. Pushdown control-flow analysis of higher-order programs. In *Workshop on Scheme and Functional Programming*, 2010.

4 M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 1992.

5 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, 1993.

6 Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *Foundations of Software Engineering*, 1995.

7 Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *POPL '95*, 1995.

8 J. Ian Johnson, Ilya Sergey, Christopher Earl, Matthew Might, and David Van Horn. Pushdown flow analysis with abstract garbage collection. *JFP*, 2014.

9 John Kodumal and Alex Aiken. The set constraint/CFL reachability connection in practice. In *PLDI*, 2004.

10 John Lamping. An algorithm for optimal lambda calculus reduction. In *POPL*, 1990.

11 Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. An incremental points-to analysis with CFL-reachability. In *Compiler Construction*, 2013.

12 Jan Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 2012.

13 Matthew Might. Abstract interpreters for free. In *Proceedings of the 17th International Conference on Static Analysis*, 2010.

14 Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *ICFP*, Portland, Oregon, 2006.

15 Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *PLDI*, 2010.

16 Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

17 Zachary Palmer and Leandro Facchinetti. Odefa proof-of-concept implementation. `https://github.com/JHU-PL-Lab/odefa-proof-of-concept`, 2015. Accessed: 2015-10-11.

18 Jakob Rehof and Manuel Fähndrich. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *POPL*, New York, NY, USA, 2001.

19 Thomas Reps. Demand interprocedural program analysis using logic databases. In *Application of Logic Databases*, 1994.

20 Thomas Reps. Shape analysis as a generalized path problem. In *PEPM*, 1995.

21 Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

22 Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991. TR CMU-CS-91-145.

23 Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In *Datalog Reloaded: First International Workshop*, 2011.

24 David Van Horn and Harry G. Mairson. Deciding kCFA is complete for EXPTIME. In *ICFP*, 2008.

25 David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP*, 2010.

26 Dimitrios Vardoulakis and Olin Shivers. CFA2: A context-free approach to control-flow analysis. In *European Symposium on Programming*, 2010.

27 Dimitrios Vardoulakis and Olin Shivers. Pushdown flow analysis of first-class control. In *ICFP*, 2011.

28 Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda-calculus*. PhD thesis, University of Oxford, 1971.

29 Xin Zhang, Mayur Naik, and Hongseok Yang. Finding optimum abstractions in parametric dataflow analysis. In *PLDI*, 2013.

Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics*

Casper Bach Poulsen[1], Pierre Néron[2], Andrew Tolmach[†3], and Eelco Visser[4]

1 **Delft University of Technology, The Netherlands**
`c.b.poulsen@tudelft.nl`
2 **French Network and Information Security Agency (ANSSI), France**
`pierre.neron@ssi.gouv.fr`
3 **Portland State University, USA**
`tolmach@pdx.edu`
4 **Delft University of Technology, The Netherlands**
`visser@acm.org`

Abstract

Semantic specifications do not make a systematic connection between the names and scopes in the static structure of a program and memory layout, and access during its execution. In this paper, we introduce a systematic approach to the alignment of names in static semantics and memory in dynamic semantics, building on the scope graph framework for name resolution. We develop a uniform memory model consisting of frames that instantiate the scopes in the scope graph of a program. This provides a language-independent correspondence between static scopes and run-time memory layout, and between static resolution paths and run-time memory access paths. The approach scales to a range of binding features, supports straightforward type soundness proofs, and provides the basis for a language-independent specification of sound reachability-based garbage collection.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Dynamic semantics, scope graphs, memory layout, type soundness, operational semantics

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.20

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.2.1.10`

1 Introduction

Name binding and memory management are pervasive concerns in programming language design. There is clearly a connection between the names and scopes in the static structure of a program and patterns of memory allocation, access, and deallocation during its execution. However, existing semantic specifications of programming languages do not treat this connection systematically, and take a wide variety of approaches to handling name binding and

* This research was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206).
† Andrew Tolmach was partly supported by a Digiteo Chair at Laboratoire de Recherche en Informatique, Université Paris-Sud.

memory management. For example, different type soundness proofs for Java-like languages might model types and memory using simple states that map identifiers and references to values [3], untyped frames that rely on traditional environments for typing [24], or ad-hoc lookup functions (or visibility predicates) designed to resolve particular kinds of identifiers such as classes or fields [6, 9]. Existing dynamic semantic specification frameworks (Redex [8], Ott [21], K [20], funcons [2]) provide little guidance in formalizing the connection between static names and dynamic memory.

In the *language designer's workbench* project [27], we are pursuing high-level declarative language specifications that are amenable to both verification and implementation using language-independent techniques. A particular goal is to separate the specifications into distinct aspects so far as possible, such as syntax, binding, typing, run-time behavior, etc. In this context we have developed *scope graphs* [14, 25] to provide a language-independent framework for static name binding and name resolution. The framework holds promise for dealing with name binding in a uniform way that scales to many binding-aware tools, such as editor services, refactoring transformations, type checkers, interpreters, and compilers.

In this paper, we address the run-time aspect of specification, and develop the relationship between static scope graphs and the *dynamic* semantics of name binding and run-time memory. Specifically, we formalize a language-independent correspondence between static scopes and *heap-allocated frames*. The approach is designed to support both verification of meta-theoretic properties and generation of realistic interpreters using fairly low-level memory operations. Our framework is *language-independent* in the sense of providing a generic set of facilities for describing and implementing binding and memory operations for a wide range of languages. We make the following contributions:

- We introduce the *scopes-as-frames* paradigm as a language-independent approach to dealing with name binding for both static and dynamic programming language semantics, by organizing frames in a run-time heap in correspondence to the organization of scopes in a scope graph, and relating static resolution paths to run-time memory access paths.
- We show how frame operations can be used in formulating big-step dynamic semantics, which resemble realistic interpreters. (The generation of high-performance frame-based interpreters from specifications in the DynSem operational semantics DSL [26] is work in progress.)
- We show how phrasing the consistency between frames and scopes as an invariant gives a language-independent principle that helps structure *type soundness* proofs by providing generic lemmas about preservation of the invariant under various memory operations.
- We demonstrate that scopes-as-frames is independent of particular language features by applying it to a functional language (Section 3), an imperative language with records (Section 4), and a class-based language with subtyping (Section 4.4). The same approach (scopes describing frames) is applied systematically to describe call frames, records, and objects instantiating a class.
- We give a high-level, language-independent specification of *sound garbage collection* of frames, and verify that several standard GC strategies refine this specification.
- We provide a Coq development[1] containing mechanized type soundness proofs for (supersets of) all languages in the paper. The type soundness proofs rely on a language-independent library formalizing scope graphs and frame heaps.

In the next section we introduce the approach by means of an example. In Section 3 we use the approach to specify and prove type soundness for a small language with first-class

[1] `https://github.com/metaborg/scopes-describe-frames/`

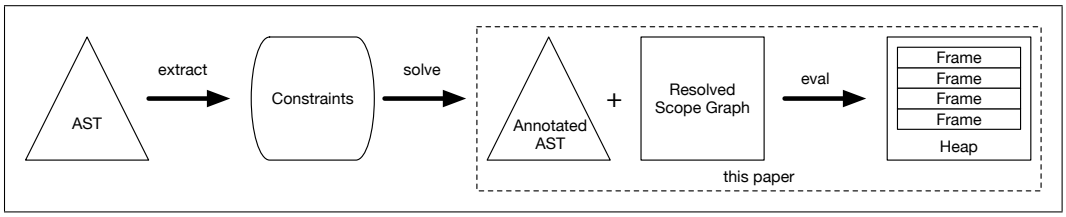


Figure 1 Architecture of the approach: static analysis of a program abstract syntax tree via constraints leads to a representation with explicit name and type information, which is the input for evaluation.

functions. In Section 4 we extend the formalization to a language with records. In Section 5 we discuss the formalization of garbage collection. In Section 6 we discuss related work. Section 7 summarizes this work and outlines ideas for future work.

2 Static Scopes Describe Dynamic Frames

Our approach builds on the theory of name resolution of Néron et al. [14, 25], which provides the foundation for a language-independent representation of static name binding in programs based on *scope graphs*. This section recalls the basics of scope graphs, and introduces a new paradigm for name binding at run time based on scope graphs.

2.1 Architecture

We first discuss how this work ties in with previous work on scope graphs. Figure 1 shows the overall architecture of the approach. We assume a program is represented by an abstract syntax tree (AST) produced by a parser. First, a set of name and type constraints is extracted from the AST. Second, a (language-independent) name and type resolution algorithm resolves the constraints and produces an annotated abstract syntax tree and a resolved scope graph [25]. Third, an interpreter evaluates this data structure using a run-time heap consisting of frames to represent the memory of the program.

This paper focuses on the third stage in Figure 1. Our starting point is a *well-bound* and *well-typed* program, represented as an *annotated AST* (where each AST node is annotated with *scope-* and *type annotations*) and a *resolved scope graph* (e.g., a scope graph produced by name and type resolution as described in [25]). Section 3 defines these notions formally. Before diving into the formal details, we illustrate the concepts of scope graphs and frame heaps by example.

2.2 Scope Graphs

Scope graphs [14, 25] provide a language-independent framework for describing static name binding and performing name resolution, which can support a wide variety of binding constructs. We recall the basic concepts of scope graphs here, starting with a simple but illustrative example.

Figure 2 presents a program that implements the factorial function and computes `fac(2)`. The figure also presents the scope graph of the program. The nodes of the scope graph represent three basic notions derived from the program abstract syntax tree: *scopes*, *declarations*, and *references*:

- A *scope* is an abstraction of a set of AST nodes that behave uniformly with respect to name binding.

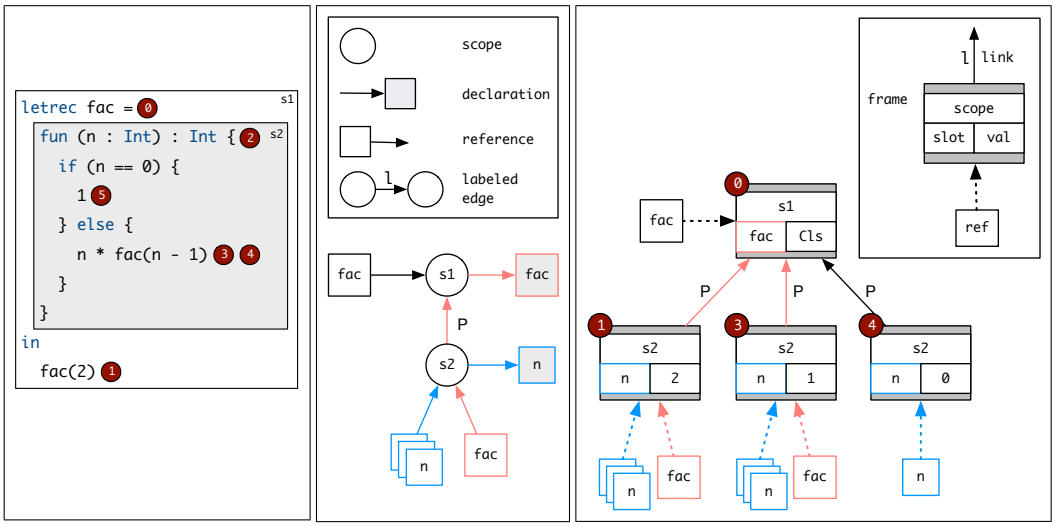


Figure 2 The left box contains an example program defining the factorial function. The boxes surrounding code blocks represent scopes. The call-outs refer to points in the execution of the program. The middle box shows the scope graph for this program; the graphical notation is explained in the legend. Distinct occurrences of similarly named references are stacked for conciseness (e.g., scope `s2` contains three distinct `n` references). The right box shows the evolution of the heap as the program executes: there is one top-level frame and three call-frames for the `fac` function.

- A *declaration* is an occurrence of an identifier that introduces a name.
- A *reference* is an occurrence of an identifier referring to a declaration.

A declaration is only visible in the scope in which it is declared and in other scopes connected to the declaration scope by a sequence of inter-scope edges. These edges represent visibility idioms such as nesting in lexical scoping and imports in module systems. A key idea is that scope graphs provide a means of resolving which declaration each reference in the AST refers to, by finding a *resolution path* from a reference to a declaration. Thus, a *reference* represents the use of a name in a scope and is *resolved* via a path along the edges of the scope graph.

In the example in Figure 2, `n` is both declared in and referenced from scope `s2`, which is associated with the function body; the blue resolution paths involve only this scope. Lexical nesting of `s2` within the outer scope `s1` is modeled by an edge labeled P (for lexical **p**arent); the reference `fac` in scope `s2` resolves to declaration `fac` in scope `s1` via the pink path.

If a given identifier is declared in more than one place, there may be multiple possible resolution paths for references to that identifier. In such cases, we choose the preferred path using a language-dependent *specificity ordering* and *path well-formedness* predicate. The resolution calculus introduced in [14] and refined in [25] defines the semantics of resolution and describes algorithms for finding most-specific resolution paths. In this paper, we assume that we already have a *resolved scope graph* in which each reference has a unique path to a corresponding declaration within the graph.

Records and field access. A key benefit of scope graphs is that they can describe many different language binding constructs in a uniform way. As a small illustration of this, we consider a language with records and named fields. Figure 3 shows an example program that defines a record type `A` with a single field `n`. The program declares two local variables, `x` and `y`. Each of these variables is first assigned new record instances of `A`; subsequently, the fields of the records stored in the variables are initialized.

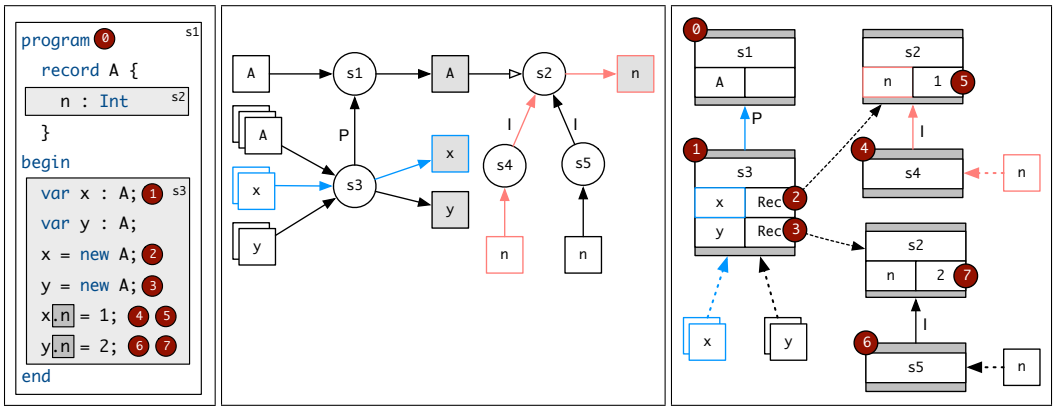


Figure 3 An example program using records (left) with its scope graph (center) and a picture of the evolution of the heap during execution (right).

The scope graph of the program in Figure 3 contains five scopes. The program scope `s1` contains the record type declaration `A`, where `A` has an *associated scope* `s2`, which is the record scope containing the record field declaration. Scope `s3` declares the local variables `x` and `y`. The two scopes `s4` and `s5` are *import scopes* containing the field references `x` and `y`, respectively. These scopes do not contain any declarations, but instead provide access to the fields of records via `I`-labeled edges. (Building these edges requires resolving the *types* of `x` and `y`; again, in this paper we assume that this resolution has already occurred, e.g. using the methods of [25].) Resolution of field names uses the same paradigm as for variables. For example, the reference to `n` in `s4` is resolved to the declaration in `s2` via the pink path.

2.3 Dynamic Frames and Heaps

We have illustrated how scope graphs provide a uniform and language-independent model for *static* name binding. In this paper, we propose a uniform and language-independent model for the layout of memory and the binding of values to names at run time. In this model, the static scope graph provides a blueprint for the layout of memory. The model is based on the notions of *frames*, *slots*, and *heaps*. Frames and heaps provide building blocks for structuring memory that are inspired by how realistic implementations of programming languages organize memory using, e.g., *stack frames* (for function calls), and *heap blocks* (for structured memory objects).

- The memory of a program at run time is a *heap* consisting of *frames*.
- A *frame* is a unit of memory consisting of *slots* mapping names to values, and *links* to other frames.
- A frame is described by a scope in the sense that each slot corresponds to a scope declaration and each link corresponds to an outgoing scope edge.
- Frames may be allocated when control passes to a program point in a new lexical scope (e.g. on function entry) or when executing an explicit memory allocation command (e.g. **new**). Frames can be garbage-collected when no longer referenced.
- The path for a reference in the static scope graph can be applied directly to fetch the referenced value at run time by interpreting it relative to the "current" frame.

The connection between scopes and frames enables us to define invariants that formally guarantee the absence of run-time errors, as Sections 3 to 5 of this paper demonstrate. In the rest of this section we illustrate how scopes-as-frames support a wide range of name binding

phenomena, including functions, let bindings, blocks with local variables, and records; all in a systematic way that follows the same pattern.

Frames and heaps for lexical scoping. Returning to the factorial function example from earlier, the rightmost illustration in Figure 2 shows the evolution of the heap as the example program executes. In this example, all the frames contain lexically-scoped variables, and correspond to stack frames in conventional language implementations. We examine the execution steps of the program to illustrate how frames and heaps are used to structure memory during run time, and how these correspond to the static scope graph.

- The program frame for scope `s1` is created at point (0); and the function closure is assigned to the `fac` slot. This is the initial current frame.
- The execution of the body of the `letrec` proceeds to initialize a new frame at point (1), which has the same structure as the scope (`s2`) that it instantiates: it is linked to a lexical parent frame that instantiates `s1`; and it has a single slot for `n`, the sole declaration of scope `s2`, which is initialized to contain `2`.
- At point (2), the newly created frame becomes the current frame. Then the statically computed resolution path for the `n` reference (colored blue in Figure 2) can be applied, starting at the current frame, to fetch the value `n` = `2`, whence a recursive call to `fac` is made in the `else` branch of the example program. Again, `fac` is dereferenced using the statically computed resolution path (highlighted in pink in Figure 2).
- The recursive call at point (3) creates another frame, and results in another recursive call and frame at point (4). At this point, `n` evaluates to `0`, and the recursive calls return at point (5).

At each step of execution, references are dereferenced relative to the current frame. The run-time memory access path for each reference is the same as its static resolution path.

Records and field access. Just as scopes provide a static model for many kinds of binding, frames support a wide range of patterns for binding at run time (not just stack-like binding). In particular, frames can be used to model structured memory objects, such as the records in our second example. These frames are allocated by an explicit **`new`** operator in the language.

The rightmost illustration in Figure 3 shows the evolution of the heap during program evaluation. Step-by-step:

- The program frame is created at point (0). This frame has a single slot for `A`, matching the declaration in scope `s1`.
- At point (1), the frame for the local variables `x` and `y` is created.
- Points (2) and (3) populate the slots for `x` and `y` with record values which point to freshly allocated record frames (indicated by the dotted lines at points (2) and (3) in Figure 3). These record frames instantiate the scope (`s2`) associated to the declaration of record type `A`.
- Next, the field slots of the record frames are filled in. The slot names are declared in record scope `s2`, but references to them are placed in intermediate scopes `s4` and `s5`, which import `s2`. To maintain a consistent relationship between static and run-time paths, we create an intermediate frame corresponding to `s4` at point (4). Then at point (5) we use the static path for `n` to locate the relevant slot in the record frame for `x` and set the value of `n` to `1`. Points (6) and (7) proceed similarly to (4) and (5).

Once more, the static and run-time resolution paths coincide, as illustrated by the colored paths in the scope graph and heap frame in Figure 3. Note that maintaining this correspondence leads to some infelicities in the dynamic frame layout. The slot created for `A` in

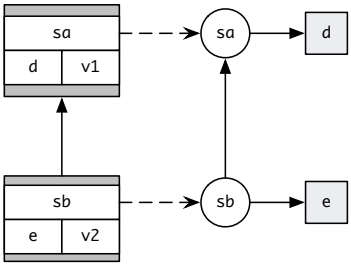


Figure 4 Frames instantiate scopes.

the program frame is actually unused and could have been omitted; declarations of purely static things, like the type name `A`, do not need a run-time correlative. Similarly, the frames corresponding to the declaration-free scopes `s4` and `s5` are empty, and having to traverse them to reach the record frame is inefficient. We could avoid both of these problems in a real system by using a more nuanced definition of correspondence between static and run-time paths; in this paper, we have elected to keep the scopes-as-frames correspondence as simple as possible.

Invariants. We have sketched how the layout of memory in frames follows the scope graph, and how this applies uniformly to a range of binding patterns. We can phrase the systematic correspondence between static scopes and dynamic frames as an invariant. Doing so gives a basis for statically checking and ruling out run-time errors. In a nutshell, the scopes and frames considered in this section all satisfy invariants that can be summarized as:

- *Binding invariant:* (a) the slots of a frame are in one-to-one correspondence with the declarations in the frame's scope; (b) the links of a frame are in one-to-one correspondence with the labeled edges of the frame's scope, and the scope of each link target matches the corresponding edge target. Figure 4 illustrates the resulting commuting diagram.
- *Typing invariant:* assuming declarations and values are typed, each slot value has the same type as its corresponding declaration.

These invariants extend to heaps by requiring that all frames in a heap satisfy the invariant. A consequence of these invariants is that each static resolution path can be used to perform corresponding run-time lookups, and (again assuming everything is typed) the values obtained from such lookups will be correctly typed.

In the next sections we formalize the framework and invariants outlined above. Concretely, we show how to define the static and dynamic semantics of a number of model languages with typical programming language binding patterns, and demonstrate and discuss how the scopes-as-frames invariant provides a basis for straightforward type soundness proofs and sound garbage collection.

3 Dynamic Frames for a Simple Functional Language

In this section we formalize our approach using L1, a language with arithmetic expressions and first-class, simply-typed functions, as a running example (Figure 8). We first present the language-independent framework of scope graphs to represent the (resolved) name binding and type facts of programs, and we formalize well-bound and well-typed L1 programs in terms of scope graphs. Then we introduce a language-independent framework of frames and heaps for modeling memory layout, and formalize the dynamic semantics of L1 in terms of it.

Scope graph

$$
\begin{array}{rcl}
\mathit{ScopeId} \ni s & & \\
\mathit{Vertex} \ni v & ::= & s \mid x_i^{\mathsf{D}} \mid x_i^{\mathsf{R}} \\
\mathit{Edge} \ni e & ::= & s \xrightarrow{l} s \mid s \longrightarrow x_i^{\mathsf{D}} \\
& \mid & x_i^{\mathsf{R}} \longrightarrow s \mid x_i^{\mathsf{D}} \longrightarrow\!\triangleright s \\
\mathit{Label} \ni l & ::= & \mathbf{P} \mid \mathbf{I} \\
\mathit{TypeAnn} \ni ta & ::= & x_i^{\mathsf{D}} : t \\
\mathcal{G} \in \mathit{ScopeGraph} & \triangleq & \wp(\mathit{Vertex}) \times \wp(\mathit{Edge}) \times \\
& & \wp(\mathit{TypeAnn})
\end{array}
$$

Projection functions

$$
\begin{array}{l}
\mathcal{K}(s) = \{l \mapsto \{s' \mid s \xrightarrow{l} s'\}\} \\
\mathcal{D}(s) = \{x_i^{\mathsf{D}} \mid s \longrightarrow x_i^{\mathsf{D}}\} \\
\mathcal{R}(s) = \{x_i^{\mathsf{R}} \mid x_i^{\mathsf{R}} \longrightarrow s\}
\end{array}
$$

Figure 5 Scope graphs.

Resolution paths

$$\mathit{Path} \ni p ::= \mathbf{D}(x_i^{\mathsf{D}}) \mid \mathbf{E}(l, s) \cdot p$$

Path consistency

$$\frac{\vdash_{\mathcal{G}} s \longrightarrow x_i^{\mathsf{D}}}{\vdash_{\mathcal{G}} \mathbf{D}(x_i^{\mathsf{D}}) : s \overset{\mathsf{S}}{\longmapsto} (s, x_i^{\mathsf{D}})} \quad [\mathsf{ResD}]$$

$$\frac{\vdash_{\mathcal{G}} s \xrightarrow{l} s' \quad \vdash_{\mathcal{G}} p : s' \overset{\mathsf{S}}{\longmapsto} (s'', x_i^{\mathsf{D}})}{\vdash_{\mathcal{G}} \mathbf{E}(l, s') \cdot p : s \overset{\mathsf{S}}{\longmapsto} (s'', x_i^{\mathsf{D}})} \quad [\mathsf{ResE}]$$

$$\frac{\vdash_{\mathcal{G}} x_i^{\mathsf{R}} \longrightarrow s \quad \vdash_{\mathcal{G}} p : s \overset{\mathsf{S}}{\longmapsto} (s', x_j^{\mathsf{D}})}{\vdash_{\mathcal{G}} p : x_i^{\mathsf{R}} \overset{\mathsf{R}}{\longmapsto} (s', x_j^{\mathsf{D}})} \quad [\mathsf{ResR}]$$

Figure 6 Resolution paths.

Finally, we will see how phrasing the consistency between frames and scopes as an invariant gives a language-independent principle for proving type soundness, which we apply to prove the type soundness of L1.

3.1 Scope Graphs

We use scope graphs [14] extended with types [25] to provide a uniform (language-independent) treatment of name binding and type assignment in the definition of the static semantics of programming languages. We introduced scope graphs by example, including their graphical notation, in the previous section. Here we formalize scope graphs and resolution.

Vertices and edges. Figure 5 formally defines what we mean by a *scope graph*, consisting of vertices and edges. A scope graph vertex is either a scope (s), a declaration (x_i^{D}), or a reference (x_i^{R}). We assume that each scope identifier, declaration, and reference is *unique* in the graph. For a declaration x_i^{D} there is exactly one scope s which contains the declaration, represented as an edge $s \longrightarrow x_i^{\mathsf{D}}$ in the scope graph; similarly, for a reference x_i^{R} there is exactly one scope s which contains the reference, represented as an edge $x_i^{\mathsf{R}} \longrightarrow s$.

In addition to edges relating scope identifiers to references or declarations, there are labeled edges connecting two scopes. A labeled edge $s \xrightarrow{\mathbf{P}} s'$ means that s has s' as its *lexical parent*, and an edge $s \xrightarrow{\mathbf{I}} s'$ means that s *imports* s'. For language L1, we only need lexical scoping, but in Section 4 we consider a language with records which relies on imports.

The bottom of Figure 5 defines some useful projection functions for a given scope graph: $\mathcal{K}(s)$ is a map from labels l to the set of scopes to which s is connected via label l; $\mathcal{D}(s)$ is the set of all declarations contained in a scope; and $\mathcal{R}(s)$ is the set of all references contained in a scope.

The last kind of edge defined in Figure 5 is an *associated scope* edge: $x_i^{\mathsf{D}} \longrightarrow\!\triangleright s$ connects the declaration x_i^{D} of a named collection of names (e.g., a module or a record) to the scope s which declares the constituent names (e.g., the body of a module or a record). Again, language L1 does not rely on associated scopes, but the language in Section 4 does.

Finally, in a *typed* scope graph, declarations are annotated with a type ($x_i^{\mathsf{D}} : t$).

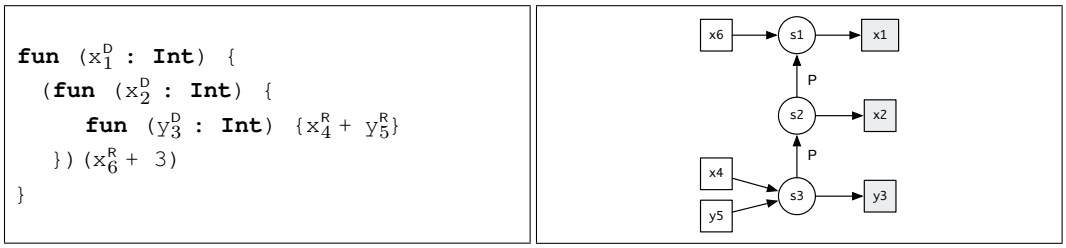


Figure 7 Example program with nested functions and its scope graph.

Resolution paths. Figure 6 defines *resolution paths*. We use the $\vdash_{\mathcal{G}}$ turnstile for predicates and relations that query an underlying scope graph $\mathcal{G}$. A path p is interpreted relative to an initial scope, and is given by a sequence of resolution steps. The resolution step $\mathbf{D}(x_i^{\mathrm{D}})$ checks that the declaration x_i^{D} is available in the initial scope, while $\mathbf{E}(l, s)$ checks that resolution continues by following the edge labeled by l from the initial scope to scope s. The path consistency relation $\vdash_{\mathcal{G}} p : s \overset{\mathrm{S}}{\longmapsto} (s', x_i^{\mathrm{D}})$ checks that a path p is consistent with the underlying scope graph, i.e. that there is an actual path in the graph from scope s to declaration x_i^{D} in scope s'. The relation $\vdash_{\mathcal{G}} p : x_i^{\mathrm{R}} \overset{\mathrm{R}}{\longmapsto} (s', x_j^{\mathrm{D}})$ checks that there is a path p from the initial scope that contains x_i^{R} to a scope s' containing x_j^{D}.

In this paper, we assume that scope graphs have been resolved, following the resolution calculus of [14, 25], so that each reference is associated with a *unique and consistent* path to a declaration. In other words, a *resolved* scope graph includes a mapping from each reference x_i^{R} to a path p which corresponds to an actual resolution, i.e. there exist s' and x_j^{D} such that $\vdash_{\mathcal{G}} p : x_i^{\mathrm{R}} \overset{\mathrm{R}}{\longmapsto} (s', x_j^{\mathrm{D}})$. When there are multiple possible resolution paths for a reference, the resolution calculus uses language-specific well-formedness and specificity ordering rules to choose which path to prefer. Since we are assuming resolution has already occurred, we ignore the details of these rules in this paper.

An example scope graph. Figure 7 defines the scope graph for an L1 expression which nests three functions (where we have replaced identifiers with unique references and declarations). Here, the $\xrightarrow{\mathbf{P}}$ edges connect the inner scope of each function to its enclosing (lexical) scope. For example, the reference $\mathrm{x}_6^{\mathrm{R}}$ has a trivial path in the scope graph. We write $p_6 = \mathbf{D}(\mathrm{x}_1^{\mathrm{D}}) : \mathrm{x}_6^{\mathrm{R}} \longmapsto \mathrm{x}_1^{\mathrm{D}}$ for this resolution path, which says that the declaration $\mathrm{x}_1^{\mathrm{D}}$ is found in the same scope as the reference $\mathrm{x}_6^{\mathrm{R}}$. Similarly, the resolution path for $\mathrm{y}_5^{\mathrm{R}}$ is $p_5 = \mathbf{D}(\mathrm{y}_3^{\mathrm{D}}) : \mathrm{y}_5^{\mathrm{R}} \longmapsto \mathrm{y}_3^{\mathrm{D}}$. In contrast, there are two possible paths from reference $\mathrm{x}_4^{\mathrm{R}}$ to matching declarations: $p_4 = \mathbf{E}(\mathbf{P}, s_2) \cdot \mathbf{D}(\mathrm{x}_2^{\mathrm{D}})$ and $p_4' = \mathbf{E}(\mathbf{P}, s_2) \cdot \mathbf{E}(\mathbf{P}, s_1) \cdot \mathbf{D}(\mathrm{x}_2^{\mathrm{D}})$. In this case, the shorter path p_4 would be preferred over the longer path p_4' according to L1's path specificity ordering rules, which encode lexical scoping (shadowing). Thus, the complete resolution mapping for this graph is $\{\mathrm{x}_6^{\mathrm{R}} \mapsto p_6, \mathrm{y}_5^{\mathrm{R}} \mapsto p_5, \mathrm{x}_4^{\mathrm{R}} \mapsto p_4\}$.

3.2 Well-Bound and Well-Typed Terms

Scope graphs provide a language-independent data structure for representing name binding and typing facts about the declarations and references in a program. To reason about the relation of this model to the program that it represents, we define a well-boundness and a well-typedness predicate on abstract syntax trees annotated with scope and type information. Figure 9 defines annotated terms for L1, ranged over by a, where expressions are annotated by their scope s and their type t. Identifiers have been replaced with unique references and declarations. We define well-boundness and well-typedness as separate predicates that check

$$
\begin{array}{l}
t ::= \texttt{Int} \mid t \to t \\
e ::= z \mid x \mid e \oplus e \mid \texttt{fun}(x : t)\{\ e\ \} \mid e(e) \\
z \in \mathbb{Z}
\end{array}
$$

Figure 8 Syntax of L1.

$$
\begin{array}{l}
t ::= \texttt{Int} \mid t \to t \\
e ::= z \mid x_i^{\mathrm{R}} \mid a \oplus a \mid \texttt{fun}(x_i^{\mathrm{D}} : t)\{\ a\ \} \mid a(a) \\
a ::= e^{s,t}
\end{array}
$$

Figure 9 Annotated syntax of L1.

Well-boundness $\boxed{\vdash^{\mathrm{B}} a}$

$$\vdash^{\mathrm{B}} z^s \quad [\mathsf{WbInt}]$$

$$\frac{\vdash_{\mathcal{G}} x_i^{\mathrm{R}} \longrightarrow s}{\vdash^{\mathrm{B}} (x_i^{\mathrm{R}})^s} \quad [\mathsf{WbRef}]$$

$$\frac{\vdash^{\mathrm{B}} e_1^s \quad \vdash^{\mathrm{B}} e_2^s}{\vdash^{\mathrm{B}} (e_1^s \oplus e_2^s)^s} \quad [\mathsf{WbArithOp}]$$

$$\frac{\vdash^{\mathrm{B}} e^{s_1} \quad \vdash_{\mathcal{G}} s_1 \xrightarrow{\mathbf{P}} s_2 \quad \vdash_{\mathcal{G}} \mathcal{D}(s_1) = \{x_i^{\mathrm{D}}\}}{\vdash^{\mathrm{B}} (\texttt{fun}\ (x_i^{\mathrm{D}} : t)\ \{\ e^{s_1}\ \})^{s_2}} \quad [\mathsf{WbFun}]$$

$$\frac{\vdash^{\mathrm{B}} e_1^s \quad \vdash^{\mathrm{B}} e_2^s}{\vdash^{\mathrm{B}} (e_1^s(e_2^s))^s} \quad [\mathsf{WbApp}]$$

Figure 10 Well-boundness of L1 terms.

Well-typedness $\boxed{\vdash^{\mathrm{T}} a}$

$$\vdash^{\mathrm{T}} z^{\texttt{Int}} \quad [\mathsf{WtInt}]$$

$$\frac{\vdash_{\mathcal{G}} p : x_i^{\mathrm{R}} \overset{\mathrm{R}}{\longmapsto} (s', x_j^{\mathrm{D}}) \quad \vdash_{\mathcal{G}} x_j^{\mathrm{D}} : t}{\vdash^{\mathrm{T}} (x_i^{\mathrm{R}})^t} \quad [\mathsf{WtRef}]$$

$$\frac{\vdash^{\mathrm{T}} e_1^{\texttt{Int}} \quad \vdash^{\mathrm{T}} e_2^{\texttt{Int}}}{\vdash^{\mathrm{T}} \left(e_1^{\texttt{Int}} \oplus e_2^{\texttt{Int}}\right)^{\texttt{Int}}} \quad [\mathsf{WtArithOp}]$$

$$\frac{\vdash_{\mathcal{G}} x_i^{\mathrm{D}} : t_1 \quad \vdash^{\mathrm{T}} e^{t_2}}{\vdash^{\mathrm{T}} (\texttt{fun}(x_i^{\mathrm{D}} : t_1)\{\ e^{t_2}\ \})^{t_1 \to t_2}} \quad [\mathsf{WtFun}]$$

$$\frac{\vdash^{\mathrm{T}} e_1^{t_1 \to t_2} \quad \vdash^{\mathrm{T}} e_2^{t_1}}{\vdash^{\mathrm{T}} \left(e_1^{t_1 \to t_2}(e_2^{t_1})\right)^{t_2}} \quad [\mathsf{WtApp}]$$

Figure 11 Well-typedness of L1 terms.

the consistency of (the annotations of) a term with respect to its scope graph. In definitions of these predicates, we omit the annotation that is not relevant for the property checked by the rule; that is, we leave out the type annotation in the well-boundness rules, and the scope annotation in the well-typedness rules. All rules are implicitly parameterized by a common underlying scope graph $\mathcal{G}$; all occurrences of $\vdash_{\mathcal{G}}$ query this $\mathcal{G}$.

The *well-boundness* rules in Figure 10 check that a term is *well-bound* relative to a resolved scope graph $\mathcal{G}$. That is, the scope graph is consistent with the scoping patterns of the term, and each reference is bound, i.e., resolves to a declaration with a path that is consistent with the scope graph. Rules [WbArithOp, WbApp] check that the child terms of arithmetic operator applications and function applications reside in the same scope as the parent term. Rule [WbRef] checks that the reference x_i^{R} is declared in the scope s of its annotation ($\vdash_{\mathcal{G}} x_i^{\mathrm{R}} \longrightarrow s$). Rule [WbFun] checks that the scope s_1 of the body of a function is a lexical child of the scope s_2 of the function expression, and that the set of declarations of the function scope is exactly the singleton set containing the formal parameter of the function.

The *well-typedness* predicate on annotated terms checks that (1) expressions are consistently typed according to the rules of the language, and (2) that each reference is typed consistently with its declaration. Figure 11 specifies well-typedness rules for L1. The Rules [WtInt, WtArithOp, WtFun, WtApp] check that the type of an expression is consistent with the types of its direct sub-expressions. Note the absence of environment threading in the well-typedness rules; scope management is taken care of by the well-boundness rules.

Frames and heaps

$$f \in \mathit{FrameId} = \{f_1, f_2, \ldots\}$$
$$ks \in \mathit{DynLinks} = \mathit{Label} \xrightarrow{\text{fin}} \mathit{ScopeId} \xrightarrow{\text{fin}} \mathit{FrameId}$$
$$\sigma \in \mathit{Slots} = \mathit{Decl} \xrightarrow{\text{fin}} \mathit{Val}$$
$$\langle s, ks, \sigma \rangle \in \mathit{Frame} = \mathit{ScopeId} \times \mathit{DynLinks} \times \mathit{Slots}$$
$$h \in \mathit{Heap} = \mathit{FrameId} \xrightarrow{\text{fin}} \mathit{Frame}$$

Projection functions

$$\mathcal{S}_h(f) = s \quad \text{where } h(f) = \langle s, ks, \sigma \rangle$$
$$\mathcal{K}_h(f) = ks \text{ where } h(f) = \langle s, ks, \sigma \rangle$$
$$\mathcal{D}_h(f) = \sigma \quad \text{where } h(f) = \langle s, ks, \sigma \rangle$$

Operations on frames

$\boxed{\mathsf{initFrame}(s, ks, \sigma)/h \Rightarrow f'/h'}$

$$\frac{f' \notin \mathit{Dom}(h)}{\mathsf{initFrame}(s, ks, \sigma)/h \Rightarrow f'/h[f' \mapsto \langle s, ks, \sigma \rangle]} \quad [\mathsf{InitFrame}]$$

Dynamic address lookup

$\boxed{\mathsf{lookup}(f, h, p) \Rightarrow \mathsf{Addr}(f', x_i^{\mathsf{D}})}$

$$\frac{x_i^{\mathsf{D}} \in \mathit{Dom}(\mathcal{D}_h(f))}{\mathsf{lookup}(f, h, \mathbf{D}(x_i^{\mathsf{D}})) \Rightarrow \mathsf{Addr}(f, x_i^{\mathsf{D}})} \quad [\mathsf{DLookupD}]$$

$$\frac{\mathcal{K}_h(f)(l)(s) = f' \quad \mathsf{lookup}(f', h, p) \Rightarrow \mathsf{Addr}(f'', x_i^{\mathsf{D}})}{\mathsf{lookup}(f, h, \mathbf{E}(l, s) \cdot p) \Rightarrow \mathsf{Addr}(f'', x_i^{\mathsf{D}})} \quad [\mathsf{DLookupE}]$$

Fetching and mutating slot values

$\boxed{\mathsf{get}(f, h, x_i^{\mathsf{D}}) \Rightarrow v}$ $\boxed{\mathsf{set}(f, h, x_i^{\mathsf{D}}, v) \Rightarrow h'}$

$$\frac{\mathcal{D}_h(f) = \sigma \quad \sigma(x_i^{\mathsf{D}}) = v}{\mathsf{get}(f, h, x_i^{\mathsf{D}}) \Rightarrow v} \quad [\mathsf{GetSlot}]$$

$$\frac{x_i^{\mathsf{D}} \in \mathit{Dom}(\mathcal{D}_h(f))}{\mathsf{set}(f, h, x_i^{\mathsf{D}}, v) \Rightarrow h[f \mapsto (\mathcal{D}_h(f)[x_i^{\mathsf{D}} \mapsto v])]} \quad [\mathsf{SetSlot}]$$

Figure 12 A language-independent formalization of frames and heaps.

Values

$$\mathit{Val} \ni v ::= \mathsf{NumV}(z) \mid \mathsf{ClosV}(x_i^{\mathsf{D}}, a, f)$$

Annotation projections

$$\mathcal{S}(e^{s,t}) = s \qquad \mathcal{T}(e^{s,t}) = t$$

Dynamic semantics

$\boxed{f \vdash a/h \Rightarrow v/h'}$

$$f \vdash z/h \Rightarrow \mathsf{NumV}(z)/h \quad [\mathsf{EvInt}]$$

$$\frac{\vdash_{\mathcal{G}} p : x_i^{\mathsf{R}} \xrightarrow{\mathsf{R}} (s', x_j^{\mathsf{D}}) \quad \mathsf{lookup}(f, h, p) \Rightarrow \mathsf{Addr}(f', x_j^{\mathsf{D}}) \quad \mathsf{get}(f', h, x_j^{\mathsf{D}}) \Rightarrow v}{f \vdash x_i^{\mathsf{R}}/h \Rightarrow v/h} \quad [\mathsf{EvRef}]$$

$$\frac{f \vdash a_1/h_1 \Rightarrow \mathsf{NumV}(z_1)/h_2 \quad f \vdash a_2/h_2 \Rightarrow \mathsf{NumV}(z_2)/h_3}{f \vdash a_1 \oplus a_2/h_1 \Rightarrow \mathsf{NumV}(\oplus(z_1, z_2))/h_3} \quad [\mathsf{EvArithOp}]$$

$$f \vdash \texttt{fun}(x_i^{\mathsf{D}} : t)\{\ a\ \}/h \Rightarrow \mathsf{ClosV}(x_i^{\mathsf{D}}, a, f)/h \quad [\mathsf{EvFun}]$$

$$\frac{\begin{array}{l} f \vdash a_1/h_1 \Rightarrow \mathsf{ClosV}(x_i^{\mathsf{D}}, a', f')/h_2 \quad f \vdash a_2/h_2 \Rightarrow v/h_3 \\ \mathsf{initFrame}(\mathcal{S}(a'), \{\mathbf{P} \mapsto \{\mathcal{S}_{h_3}(f') \mapsto f'\}\}, \{x_i^{\mathsf{D}} \mapsto v\})/h_3 \Rightarrow f''/h_4 \\ f'' \vdash a'/h_4 \Rightarrow v'/h_5 \end{array}}{f \vdash a_1(a_2)/h_1 \Rightarrow v'/h_5} \quad [\mathsf{EvApp}]$$

Figure 13 Dynamic semantics of L1.

3.3 Frames and Heaps

We now develop the formalization of the language-independent system of heaps and frames to represent memory at run-time. Figure 12 defines frames, heaps, and some operations on them that we will use in the dynamic semantics of L1. The framework is parameterized with a domain *Val* of values.

A heap h is a finite map from frame identifiers (pointers) to frames. A frame consists of a scope identifier, a collection of dynamic links, and a collection of slots. The scope identifier refers to the scope that the frame instantiates. The dynamic links are defined as a two-dimensional mapping from labels and scopes to frame identifiers. These links are the dynamic counterparts of labeled edges in the scope graph and implement the link from a frame to the frame that represents its lexical context or an imported module. The slots of a frame are defined as a finite map from declarations to values.

We define operations on heaps to construct new frames and retrieve slot values. Note that all operations will refer to frames using their frame identifier (f). Accessing frames is done via a lookup in the heap ($h(f)$) and using the projection functions $\mathcal{S}\_(\_)$, $\mathcal{K}\_(\_)$, or $\mathcal{D}\_(\_)$. We will write 'frame f' as shorthand for 'the frame referred to by frame identifier f'. The $\mathsf{initFrame}(s, ks, \sigma)$ operation adds a new frame with scope identifier s, links ks, and slots σ, to the heap at a fresh frame identifier. The operation is defined as a relation $\mathsf{initFrame}(s, ks, \sigma)/h \Rightarrow f'/h'$, using the notation t/h to represent the pair of a term (or value) t and a heap h. We use this notation throughout to explicitly represent heap threading. (We could use a state monad to implicitly thread the heap, but we prefer to make semantic rules explicit for this presentation.)

The lookup operation $\mathsf{lookup}(f, h, p) \Rightarrow \mathsf{Addr}(f', x_i^{\mathrm{D}})$ dereferences a path p relative to frame f, resulting in an *address*, consisting of a pair of a frame f' and a slot x_i^{D} in that frame. The operation $\mathsf{get}(f, h, x_i^{\mathrm{D}}) \Rightarrow v$ retrieves the value v of slot x_i^{D} from frame f; and $\mathsf{set}(f, h, x_i^{\mathrm{D}}, v) \Rightarrow h'$ sets the value of slot x_i^{D} of frame f to v, resulting in a new heap h'.

3.4 Dynamic Semantics

Figure 13 specifies the big-step dynamic semantics for L1 using a heap to represent the memory during execution. The value domain *Val* consists of numbers $\mathsf{NumV}(z)$ and closures $\mathsf{ClosV}(x_i^{\mathrm{D}}, a, f)$. The judgment $f \vdash a/h \Rightarrow v/h'$ specifies that evaluating an annotated expression a in the context of frame f in heap h gives a value v and heap h'. We discuss the rules and use the example in Figure 14 as illustration:

- The Rules [EvInt,EvArithOp] for arithmetic are standard and do not affect memory other than passing the current frame to evaluation of arguments and threading the heap.
- Rule [EvFun] constructs a *closure* which records the formal parameter, the body of the function, and the lexical context at the point of function definition, which is exactly represented by the current frame f. In Figure 14, the result of evaluating the function in the body of the function at (3) leads to a closure with frame `s3` as lexical context.
- Rule [EvApp] defines the evaluation of a function application. The function argument a_1 should evaluate to a closure. Using initFrame a new frame instantiating the scope of the function body ($\mathcal{S}(a')$) is constructed to represent the call frame of the function call. The frame from the closure (f') is used as the lexical parent of the call frame. The argument value (v) is assigned to the slot for the formal parameter of the function (x_i^{D}). The body of the function (a') is evaluated in the context of the new frame. In Figure 14, in the application `f(1)`, `f` evaluates to a closure with a pointer to the `s3` frame. The call frame for the body then constructs a frame for `s4` with the `s3` frame as its parent.

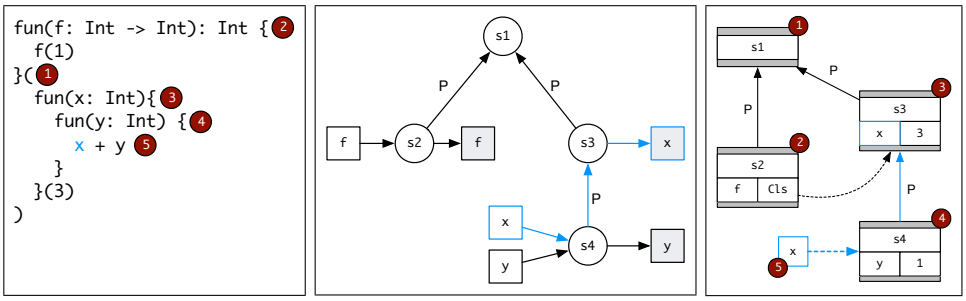


Figure 14 Example L1 program with scope graph and heap.

Rule [EvRef] defines the evaluation of a variable x_i^{R}. The frame f represents the lexical context of the variable and the static scope graph path of the (reference representing the) variable is the offset into that context. The lookup happens in multiple steps: first, we find the path associated with x_i^{R} in the resolved scope graph; second, we dynamically follow the path to compute the dynamic memory address for the reference; lastly, we get the value of the dynamic memory address. In Figure 14, the evaluation at (5) of variable `x` in the `s4` frame follows the path defined for `x` in the scope graph. (Note how the call-time context of `f` is not accessible to the evaluation of the closure in frame `s4`.)

3.5 Intermezzo

What have we learned from this exercise so far? (1) *There is a systematic correspondence between static name binding and binding at run-time.* Static scopes are collections of declarations. Dynamic frames are units of memory allocation, holding values for the declarations in a scope. Static name binding is about visibility of declarations following a path from reference through the scope graph. Binding at run-time is about paths from evaluation frame to storage frame. This correspondence provides a guiding principle for the definition of the dynamic semantics. Evaluating a binding construct requires creation of a frame that instantiates the corresponding static scope. Evaluating a reference requires looking up its value in the heap using its static path as offset. (2) *We can mostly separate the specification of binding and typing rules.* Binding rules are concerned with describing the structure of and the access to memory. Typing rules are concerned with the types of the arguments and results of operations. In the dynamic semantics this corresponds to factoring out (and making systematic and uniform) the treatment of memory from other 'domain-specific' operations (such as arithmetic). We have seen that typing depends on binding in order to establish the types of references. In the next section we will see that binding depends on typing exactly when types are used to describe memory in data type definitions.

In conclusion: defining name binding in terms of scope graphs provides a uniform methodology (an API if you want) for the treatment of memory in static and dynamic semantics. This scales to a range of language features: in Section 4 we show that the approach scales to imperative features and records, and in Section 4.4 we discuss how to deal with classes and subtyping.

What other benefits does the approach provide? The design of this uniform memory model is just the start of a study of making language-independent those aspects of programming language design that are usually treated as language-specific. In Section 3.6 we will see how phrasing the consistency between frames and scopes as an invariant gives a *language-independent principle* for proving *type soundness*, i.e. that well-bound and well-typed programs cannot go wrong. Later, in Section 5 we discuss a high-level *language-independent*

Good frame

For any definition of types t, values v, and value typing relation $\overset{\mathrm{V}}{\vdash}$:

$\mathsf{wellBound}(h, f) \triangleq$
$\quad \exists s.\ s = \mathcal{S}_h(f)\ \wedge\ \mathcal{D}(s) = \mathit{Dom}(\mathcal{D}_h(f))\ \wedge$
$\quad\quad (\forall l\ s'.\ s' \in \mathcal{K}(s)(l)\ \iff\ \mathcal{S}_h(\mathcal{K}_h(f)(l)(s')) = s')$

$\mathsf{wellTyped}(h, f) \triangleq$
$\quad \forall t\ x_i^{\mathrm{D}} \in \mathcal{D}(\mathcal{S}_h(f)).\ \vdash_{\mathcal{G}} x_i^{\mathrm{D}} : t\ \implies\ \forall v.\ \mathcal{D}_h(f)(x_i^{\mathrm{D}}) = v\ \implies\ h \overset{\mathrm{V}}{\vdash} v : t$

$\mathsf{goodFrame}(h, f) \triangleq$
$\quad \mathsf{wellBound}(h, f)\ \wedge\ \mathsf{wellTyped}(h, f)$

Good heap

$\mathsf{goodHeap}(h) \triangleq (f \in \mathit{Dom}(h)\ \implies\ \mathsf{goodFrame}(h, f))$

Value typing $\boxed{h \overset{\mathrm{V}}{\vdash} v : t}$

$h \overset{\mathrm{V}}{\vdash} \mathsf{NumV}(z) : \texttt{Int}$ [VtInt]

$$\frac{\overset{\mathrm{B}}{\vdash} \texttt{fun}(x_i^{\mathrm{D}} : t_1)\{\ a\ \}^{\mathcal{S}_h(f)} \quad \overset{\mathrm{T}}{\vdash} \texttt{fun}(x_i^{\mathrm{D}} : t_1)\{\ a\ \}^{t_1 \to t_2}}{h \overset{\mathrm{V}}{\vdash} \mathsf{ClosV}(x_i^{\mathrm{D}}, a, f) : t_1 \to t_2}\ [\mathsf{VtClo}]$$

Figure 15 Good frames, good heaps, and value typing.

specification of *sound reachability-based garbage collection* of frames, and verify that several standard GC strategies refine this specification.

3.6 Type Soundness

We have argued that the structure of frames follows the structure of scopes. Now we make this argument precise by defining the correspondence between frames and scopes, and discuss how that supports a language-independent formulation and systematic proof of type soundness.

Good frames. The goodFrame property defined in Figure 15 formally captures the correspondence between frames and scopes. The definition relies on two auxiliary properties:

1. wellBound: Maintains the binding invariant that frames have the same structure as their scopes. Specifically, the slots $\mathcal{D}_h(f)$ of each frame f are in one-to-one correspondence with the declarations of scope $s = \mathcal{S}_h(f)$, and the links $\mathcal{K}_h(f)$ of f are in one-to-one correspondence with the edges $\{l \mapsto s'\} \in \mathcal{K}(s)$, such that $\mathcal{K}_h(f)(l)(s') = f'$ iff $\mathcal{S}_h(f') = s'$.
2. wellTyped: Maintains the typing invariant that slots contain values of the type that their declarations expect. Specifically, for each declaration and its corresponding slot (if it exists) in the frame, the value in the slot is of the same type as the declaration.

These properties are language-independent, i.e. define a generic correspondence between scopes and frames, but are parameterized by a language-specific value typing judgment $h \overset{\mathrm{V}}{\vdash} v : t$ (also defined in Figure 15) which asserts that the value v has type t in the heap h. The value typing of closures (Rule [VtClo]) checks that the closure corresponds to an actual well-typed and well-bound function relative to the scope of the lexical frame recorded in the closure, using the rules from Figure 10 and Figure 11.

Good heaps. The goodHeap property also defined in Figure 15 generalizes the goodFrame property to the entire heap; that is, a heap h is good iff every frame in h satisfies the goodFrame property.

Language-independent lemmas. The scopes-as-frames approach provides structure to type soundness proofs by using a small suite of language-independent helper lemmas about dynamic memory operations, such as looking up a path, initializing a new frame, etc. For example, Lemma 1 formalizes the property that each static resolution path has a corresponding run-time memory access path.

▶ **Lemma 1** (Good paths). *Suppose* $\mathsf{goodHeap}(h)$ *and* $\mathcal{S}_h(f) = s$. *For any reference* x_i^{R} *where* $\vdash_{\mathcal{G}} x_i^{\mathrm{R}} \longrightarrow s$, *it holds for any* p, s', *and* x_i^{D} *that* $\vdash_{\mathcal{G}} p : x_i^{\mathrm{R}} \overset{\mathrm{R}}{\longmapsto} (s', x_j^{\mathrm{D}})$ *implies that there exists a frame* f' *such that* $\mathsf{lookup}(f, h, p) \Rightarrow \mathsf{Addr}(f', x_j^{\mathrm{D}})$ *and* $\mathcal{S}_h(f') = s'$.

This, along with a number of other lemmas, provides structure to the type preservation proof that we discuss next. We refer the reader to our Coq development for the full details.

Type preservation. Theorem 2 proves type preservation, i.e., that if evaluation of a well-bound and well-typed program in a frame that is in a good heap succeeds, then the resulting value is of the expected type, and the resulting heap is good.

▶ **Theorem 2** (Type preservation for L1). *Suppose* $\mathsf{goodHeap}(h_1)$ *and* $\mathcal{S}_{h_1}(f) = s$, *as well as* $\overset{\mathrm{B}}{\vdash} e^s$ *and* $\overset{\mathrm{T}}{\vdash} e^t$ *hold. Then, for any* v *and* h_2, $f \vdash e^{s,t}/h_1 \Rightarrow v/h_2$ *implies* $\mathsf{goodHeap}(h_2)$ *and* $h_2 \overset{\mathrm{V}}{\vdash} v : t$.

There are several ways to extend type preservation proofs to type soundness proofs. A traditional approach [11] to proving type soundness using big-step semantics is to add explicit "wrong" rules to a big-step semantics anywhere evaluation can get stuck. This allows distinguishing between getting stuck and diverging, which is otherwise impossible using the inductively defined big-step rules in Figure 13. Our accompanying technical report summarizes the rules that need to be added to the language. The rules and full type soundness proof can also be found in our Coq development. Adding wrong rules to a language does not alter the structure of the cases for the preservation proof. Section 6 recalls and discusses alternative means of proving type soundness using big-step semantics.

4 Dynamic Frames for Records

In this section we apply our approach to L2, an extension of L1 with records and imperative features. This shows that frames-as-scopes scales to model the memory layout for records, and that the invariants about frames and heaps given in the previous section also provide a basis for type soundness proofs for languages with such features.

Figure 16 defines the well-formed terms of L2 with changes from L1 highlighted. A program in L2 consists of a sequence of *record type definitions* d, and a program expression e, enclosed in a **`begin` ... `end`** block. A record type definition declares the type name and the record's field names. Records are constructed using the **`new`** construct, which takes a reference that resolves to a record type name declaration. L2 also adds an assignment operation (`_ := _`), which assigns a value to a memory address. Memory addresses (or *L-values* [23]) consist of frame-declaration pairs (f, x_i^{D}), and are computed by left-hand side (lhs) expressions, namely variables (x_i^{R}) and record field access $(a.x_i^{\mathrm{R}})$. Finally, L2 has a sequential composition operator (`_; _`).

4.1 Well-Bound and Well-Typed Terms

Figure 17 specifies the well-boundness rules for the newly introduced L2 terms. The well-boundness of L1 expressions from Figure 10 carry over to L2. Well-boundness checking in L2 uses three new kinds of judgments: $\vdash^{\mathrm{B}}_{\mathrm{P}}, \vdash^{\mathrm{T}}_{\mathrm{P}}$ for programs, $\vdash^{\mathrm{B}}_{\mathrm{D}}, \vdash^{\mathrm{T}}_{\mathrm{D}}$ for declarations, and $\vdash^{\mathrm{B}}_{\mathrm{L}}, \vdash^{\mathrm{T}}_{\mathrm{L}}$ for lhs expressions. The well-typedness of terms in L1 also carries over to L2. Figure 18 defines rules for checking the well-typedness of the new programs and terms in L2.

- Rule [WbProg] defines the well-boundness of programs and checks that the record definitions ds are well-bound, and that the expression e of a program is well-bound.
- Rule [WbRD] in Figure 17 checks that the type name x_i^{D} is declared in the surrounding scope s, that it is associated with scope s', which has the surrounding scope s as its lexical parent, and that each field of the record is declared in scope s'.
- Rule [WbNew] checks that the record name resolves to a record declaration. Also, the associated record scope does not have any edges, a fact which is needed when initializing a matching record frame.
- Rules [WbAsgn] and [WtAsgn] simply check that both the lhs expression and the value expression are well-bound and well-typed. Rules [WbLhs,WtLhs] in turn rely on a separate judgment for checking that an lhs expression (either a field access or a variable) is well-bound and well-typed.
- Rules [WbLFAcc] and [WtLFAcc] describe the static semantics of field access. In a field access expression such as $a\,.\,x_i^{\mathrm{R}}$, the annotated expression a computes a record, and the reference x_i^{R} refers to a field declared in this record. The field reference is installed in an auxiliary scope s' that imports the scope s_{rec} associated with the record type $\mathbf{Rec}\,(x_j^{\mathrm{D}})$.

4.2 Dynamic Semantics

The dynamic semantics of most existing constructs from L1 is unchanged in L2. Figure 19 gives the dynamic semantics of the new constructs in the L2 extension and replaces the rule for variables since variables are now lhs expressions. We describe these changes.

Rule [EvNew] says that evaluating a **new** expression constructs a record frame via $\mathsf{initDefault}(s, ks)$. Here, s is a scope identifier and ks is the map of dynamic links for the frame. As specified in Figure 20, initDefault instantiates all slots of the frame with *default values*. The $\vdash^{\mathrm{D}}_{\mathrm{V}}$ judgment in Figure 20 defines default values for all types in L2. We introduce a special default function ($\mathsf{DFun}(v)$) which, when applied to anything, returns the value v. Rule [EvAppDef] specifies the semantics for applications involving this value.

We also introduce the null value, NullV, as the default value for record types. Trying to access a NullV value as a field results in a null-pointer exception being raised (Rule [LhsFAccNull]). Not shown in Figure 19 are the straightforward rules for propagating such exceptions, which can be found in the accompanying technical report [17]. We choose to let frames contain null values by default for several reasons: firstly, null values (or a variant thereof) are required if we want to construct self-referential records. Secondly, using default values and null values ensures that frames are always well-typed, which makes reasoning about type soundness easier.

The rules in Fig. 19 closely follow the static semantics for binding and typing. For example, Rule [LhsFAcc] initializes an import frame which is used as the basis for dynamic resolution. In order to evaluate lhs expressions, a new judgment $f \vdash^{\mathrm{LHS}} lhs/h \Rightarrow u/h'$ is introduced that evaluates a *lhs* to an address, or throws a null-pointer exception if the lhs expression attempts to access a field of a null value. Lhs expressions compute auxiliary values ranged over by u, as defined in Figure 19.

$$
\begin{array}{rcl}
t & ::= & \texttt{Int} \mid t \to t \mid \texttt{Rec}\,(x_i^{\mathrm{D}}) \\
p & ::= & d^*\ \texttt{begin}\ e\ \texttt{end} \\
d & ::= & \texttt{record}\ x_i^{\mathrm{D}}\ \{\ (x_i^{\mathrm{D}}\ \texttt{:}\ t)^*\ \} \\
e & ::= & z \mid lhs \mid a \oplus a \mid \texttt{fun}\,(x_i^{\mathrm{D}}\texttt{:}\ t)\{\ a\ \} \mid a(a) \mid \texttt{new}\ x_i^{\mathrm{R}} \mid lhs\ \texttt{:=}\ a \mid a\texttt{;}\ a \\
lhs & ::= & x_i^{\mathrm{R}} \mid a\,.\,x_i^{\mathrm{R}} \\
a & ::= & e^{s,t}
\end{array}
$$

Figure 16 Annotated syntax of L2 with distinguished references and declarations.

Well-bound programs $\vdash^{\mathrm{B}}_{\mathrm{P}} p$

$$\frac{(\forall d \in ds.\ \vdash^{\mathrm{B}}_{\mathrm{D}} d^s) \quad \vdash^{\mathrm{B}} e^s}{\vdash^{\mathrm{B}}_{\mathrm{P}} ds\ \texttt{begin}\ e^s\ \texttt{end}}\ [\mathsf{WbProg}]$$

Well-bound declarations $\vdash^{\mathrm{B}}_{\mathrm{D}} d^s$

$$\frac{\vdash_{\mathcal{G}} s \longrightarrow x_i^{\mathrm{D}} \quad \vdash_{\mathcal{G}} x_i^{\mathrm{D}} \longrightarrow\!\triangleright s' \quad \forall (x_j^{\mathrm{D}}\ \texttt{:}\ t) \in \mathit{flds}.\ \vdash_{\mathcal{G}} s' \longrightarrow x_j^{\mathrm{D}}}{\vdash^{\mathrm{B}}_{\mathrm{D}} (\texttt{record}\ x_i^{\mathrm{D}}\ \{\ \mathit{flds}\ \})^s}\ [\mathsf{WbRD}]$$

Well-bound expressions $\vdash^{\mathrm{B}} e^s$

$$\frac{\vdash_{\mathcal{G}} x_i^{\mathrm{R}} \longrightarrow s \quad \vdash_{\mathcal{G}} p : x_i^{\mathrm{R}} \overset{\mathrm{R}}{\longmapsto} (s', x_j^{\mathrm{D}}) \quad \vdash_{\mathcal{G}} x_j^{\mathrm{D}} \longrightarrow\!\triangleright s_{rec} \quad \vdash_{\mathcal{G}} \mathcal{K}(s_{rec}) = \emptyset}{\vdash^{\mathrm{B}} (\texttt{new}\ x_i^{\mathrm{R}})^s}\ [\mathsf{WbNew}]$$

$$\frac{\vdash^{\mathrm{B}} lhs^s \quad \vdash^{\mathrm{B}} e^s}{\vdash^{\mathrm{B}} (lhs\ \texttt{:=}\ e^s)^s}\ [\mathsf{WbAsgn}]$$

$$\frac{\vdash^{\mathrm{B}} e_1^s \quad \vdash^{\mathrm{B}} e_2^s}{\vdash^{\mathrm{B}} (e_1^s\texttt{;}\ e_2^s)^s}\ [\mathsf{WbSeq}]$$

$$\frac{\vdash^{\mathrm{B}}_{\mathrm{L}} lhs^s}{\vdash^{\mathrm{B}} lhs^s}\ [\mathsf{WbLhs}]$$

Well-bound lhs expressions $\vdash^{\mathrm{B}}_{\mathrm{L}} lhs^s$

$$\frac{\vdash^{\mathrm{B}} e^s \quad \vdash_{\mathcal{G}} x_i^{\mathrm{R}} \longrightarrow s' \quad \vdash_{\mathcal{G}} s' \xrightarrow{\mathbf{I}} s_{rec} \quad \vdash_{\mathcal{G}} \mathcal{D}(s') = \emptyset}{\vdash^{\mathrm{B}}_{\mathrm{L}} (e^s\,.\,x_i^{\mathrm{R}})^s}\ [\mathsf{WbLFAcc}]$$

$$\frac{\vdash_{\mathcal{G}} x_i^{\mathrm{R}} \longrightarrow s}{\vdash^{\mathrm{B}}_{\mathrm{L}} (x_i^{\mathrm{R}})^s}\ [\mathsf{WbLVar}]$$

Figure 17 Well-boundness in L2.

Well-typed programs $\vdash^{\mathrm{T}}_{\mathrm{P}} p$

$$\frac{(\forall d \in ds.\ \vdash^{\mathrm{T}}_{\mathrm{D}} d) \quad \vdash^{\mathrm{T}} e^t}{\vdash^{\mathrm{T}}_{\mathrm{P}} ds\ \texttt{begin}\ e^t\ \texttt{end}}\ [\mathsf{WtProg}]$$

Well-typed declarations $\vdash^{\mathrm{T}}_{\mathrm{D}} d^t$

$$\frac{\forall (x_j^{\mathrm{D}}\ \texttt{:}\ t) \in \mathit{flds}.\ \vdash_{\mathcal{G}} x_j^{\mathrm{D}} : t}{\vdash^{\mathrm{T}}_{\mathrm{D}} \texttt{record}\ x_i^{\mathrm{D}}\ \{\ \mathit{flds}\ \}}\ [\mathsf{WtRD}]$$

Well-typed expressions $\vdash^{\mathrm{T}} e^t$

$$\frac{\vdash_{\mathcal{G}} p : x_i^{\mathrm{R}} \overset{\mathrm{R}}{\longmapsto} (s', x_j^{\mathrm{D}})}{\vdash^{\mathrm{T}} (\texttt{new}\ x_i^{\mathrm{R}})^{\texttt{Rec}\,(x_j^{\mathrm{D}})}}\ [\mathsf{WtNew}]$$

$$\frac{\vdash^{\mathrm{T}}_{\mathrm{L}} lhs^t \quad \vdash^{\mathrm{T}} e^t}{\vdash^{\mathrm{T}} (lhs\ \texttt{:=}\ e^t)^t}\ [\mathsf{WtAsgn}]$$

$$\frac{\vdash^{\mathrm{T}} e_1^{t_1} \quad \vdash^{\mathrm{T}} e_2^{t_2}}{\vdash^{\mathrm{T}} \left(e_1^{t_1}\texttt{;}\ e_2^{t_2}\right)^{t_2}}\ [\mathsf{WtSeq}]$$

$$\frac{\vdash^{\mathrm{T}}_{\mathrm{L}} lhs^t}{\vdash^{\mathrm{T}} lhs^t}\ [\mathsf{WtLhs}]$$

Well-typed lhs expressions $\vdash^{\mathrm{T}}_{\mathrm{L}} lhs^t$

$$\frac{\vdash^{\mathrm{T}} e^{\texttt{Rec}\,(x_j^{\mathrm{D}})} \quad \vdash_{\mathcal{G}} x_j^{\mathrm{D}} \longrightarrow\!\triangleright s_{rec} \quad \vdash_{\mathcal{G}} x_i^{\mathrm{R}} \longrightarrow s' \quad \vdash_{\mathcal{G}} s' \xrightarrow{\mathbf{I}} s_{rec} \quad \vdash_{\mathcal{G}} p : x_i^{\mathrm{R}} \overset{\mathrm{R}}{\longmapsto} (s'', x_i^{\mathrm{D}}) \quad \vdash_{\mathcal{G}} x_i^{\mathrm{D}} : t}{\vdash^{\mathrm{T}}_{\mathrm{L}} \left(e^{\texttt{Rec}\,(x_j^{\mathrm{D}})}\,.\,x_i^{\mathrm{R}}\right)^t}\ [\mathsf{WtLFAcc}]$$

$$\frac{\vdash_{\mathcal{G}} p : x_i^{\mathrm{R}} \overset{\mathrm{R}}{\longmapsto} (s', x_i^{\mathrm{D}}) \quad \vdash_{\mathcal{G}} x_i^{\mathrm{D}} : t}{\vdash^{\mathrm{T}}_{\mathrm{L}} (x_i^{\mathrm{R}})^t}\ [\mathsf{WtLVar}]$$

Figure 18 Well-typedness in L2.

Values

$Val \ni v ::= \ldots \mid \mathsf{RecordV}(f) \mid \mathsf{NullV} \mid \mathsf{ExcV}(X) \mid \mathsf{DFun}(v)$

$Exc \ni X ::= \mathsf{NullPointer}$

Auxiliary values

$AuxVal \ni u ::= \mathsf{Addr}(f, x_i^{\mathrm{D}}) \mid \mathsf{ExcV}(X)$

Program evaluation $\boxed{\vdash^{\mathrm{P}} p \Rightarrow v/h'}$

$$\frac{\mathsf{initDefault}(s, \emptyset)/\emptyset \Rightarrow f_{root}/h \quad f_{root} \vdash e/h \Rightarrow v/h'}{\vdash^{\mathrm{P}} ds\ \mathtt{begin}\ e^s\ \mathtt{end} \Rightarrow v/h'} \text{[EvProg]}$$

Expression evaluation $\boxed{f \vdash a/h \Rightarrow v/h'}$

$$\frac{f \vdash^{\mathrm{LHS}} lhs/h_1 \Rightarrow \mathsf{Addr}(f', x_i^{\mathrm{D}})/h_2 \quad \mathsf{get}(f', h_2, x_i^{\mathrm{D}}) \Rightarrow v}{f \vdash lhs/h_1 \Rightarrow v/h_2} \text{[EvLhs]}$$

$$\frac{\vdash_{\mathcal{G}} p : x_i^{\mathrm{R}} \overset{\mathrm{R}}{\longmapsto} (s', x_j^{\mathrm{D}}) \quad \vdash_{\mathcal{G}} x_j^{\mathrm{D}} \longrightarrow\!\!\triangleright s \quad \mathsf{initDefault}(s, \emptyset)/h_1 \Rightarrow f'/h_2}{f \vdash \mathtt{new}\ x_i^{\mathrm{R}}/h_1 \Rightarrow \mathsf{RecordV}(f')/h_2} \text{[EvNew]}$$

$$\frac{f \vdash^{\mathrm{LHS}} lhs/h_1 \Rightarrow \mathsf{Addr}(f', x_i^{\mathrm{D}})/h_2 \quad f \vdash e/h_2 \Rightarrow v/h_3 \quad \mathsf{set}(f', h_3, x_i^{\mathrm{D}}, v) \Rightarrow h_4}{f \vdash lhs\ \mathtt{:=}\ e/h_1 \Rightarrow v/h_4} \text{[EvAsgn]}$$

$$\frac{f \vdash a_1/h_1 \Rightarrow \mathsf{DFun}(v_1)/h_2 \quad f \vdash a_2/h_2 \Rightarrow v_2/h_3}{f \vdash a_1(a_2)/h_1 \Rightarrow v_1/h_3} \text{[EvAppDef]}$$

Lhs evaluation $\boxed{f \vdash^{\mathrm{LHS}} lhs/h \Rightarrow u/h'}$

$$\frac{\vdash_{\mathcal{G}} p : x_i^{\mathrm{R}} \overset{\mathrm{R}}{\longmapsto} (s', x_j^{\mathrm{D}}) \quad \mathsf{lookup}(f, h, p) \Rightarrow \mathsf{Addr}(f', x_j^{\mathrm{D}})}{f \vdash^{\mathrm{LHS}} x_i^{\mathrm{R}}/h \Rightarrow \mathsf{Addr}(f', x_j^{\mathrm{D}})/h} \text{[LhsVar]}$$

$$\frac{\begin{array}{l} f \vdash e/h_1 \Rightarrow \mathsf{RecordV}(f_{rec})/h_2 \quad \vdash_{\mathcal{G}} x_i^{\mathrm{R}} \longrightarrow\!\!\blacktriangleright s \\ \mathsf{initDefault}(s, \{\mathbf{I} \mapsto \{\mathcal{S}_{h_2}(f_{rec}) \mapsto f_{rec}\}\})/h_2 \Rightarrow f'/h_3 \\ \vdash_{\mathcal{G}} p : x_i^{\mathrm{R}} \overset{\mathrm{R}}{\longmapsto} (s', x_j^{\mathrm{D}}) \quad \mathsf{lookup}(f', h_3, p) \Rightarrow \mathsf{Addr}(f'', x_j^{\mathrm{D}}) \end{array}}{f \vdash^{\mathrm{LHS}} e.x_i^{\mathrm{R}}/h_1 \Rightarrow \mathsf{Addr}(f'', x_j^{\mathrm{D}})/h_3} \text{[LhsFAcc]}$$

$$\frac{f \vdash e/h_1 \Rightarrow \mathsf{NullV}/h_2}{f \vdash^{\mathrm{LHS}} e.x_i^{\mathrm{R}}/h_1 \Rightarrow \mathsf{ExcV}(\mathsf{NullPointer})/h_2} \text{[LhsFAccNull]}$$

Figure 19 Dynamic semantics of L2.

Frame operations $\boxed{\mathsf{initDefault}(s, ks)/h \Rightarrow f'/h'}$

$$\frac{\mathsf{defaults}(\mathcal{D}(s)) \Rightarrow \sigma \quad \mathsf{initFrame}(s, ks, \sigma)/h \Rightarrow f'/h'}{\mathsf{initDefault}(s, ks)/h \Rightarrow f'/h'} \text{[InitDefault]}$$

Default slots $\boxed{\mathsf{defaults}(ds) \Rightarrow \sigma}$

$$\mathsf{defaults}(\emptyset) \Rightarrow \emptyset \text{ [DNil]}$$

$$\frac{x_i^{\mathrm{D}} \in ds \quad \vdash_{\mathcal{G}} x_i^{\mathrm{D}} : t \quad \vdash^{\mathrm{DV}} v : t \quad \mathsf{defaults}(ds - \{x_i^{\mathrm{D}}\}) \Rightarrow \sigma}{\mathsf{defaults}(ds) \Rightarrow \sigma[x_i^{\mathrm{D}} \mapsto v]} \text{[DCons]}$$

Default values $\boxed{\vdash^{\mathrm{DV}} t : v}$

$$\vdash^{\mathrm{DV}} 0 : \mathtt{Int} \text{ [DefaultInt]}$$

$$\frac{\vdash^{\mathrm{DV}} v : t_2}{\vdash^{\mathrm{DV}} \mathsf{DFun}(v) : t_1 \to t_2} \text{[DefaultFun]}$$

$$\vdash^{\mathrm{DV}} \mathsf{NullV} : \mathtt{Rec}\,(x_i^{\mathrm{D}}) \text{ [DefaultRec]}$$

Figure 20 Frame operations for updating slots and initializing frames with default values.

$$\frac{\vdash_{\mathcal{G}} x_i^{\mathrm{D}} \longrightarrow\!\triangleright \mathcal{S}_h(f)}{h \overset{\mathrm{V}}{\vdash} \mathsf{RecordV}(f) : \mathbf{Rec}\,(x_i^{\mathrm{D}})} \quad [\mathsf{VtRec}]$$

$$\frac{h \overset{\mathrm{V}}{\vdash} v : t_2}{h \overset{\mathrm{V}}{\vdash} \mathsf{DFun}(v) : t_1 \to t_2} \quad [\mathsf{VtDefFun}]$$

$$h \overset{\mathrm{V}}{\vdash} \mathsf{NullV} : \mathbf{Rec}\,(x_i^{\mathrm{D}}) \quad [\mathsf{VtNullRec}]$$

$$h \overset{\mathrm{V}}{\vdash} \mathsf{ExcV}(\mathsf{NullPointer}) : t \quad [\mathsf{VtNullPointer}]$$

Figure 21 Value-typing of records and null-values.

Rule [EvAsgn] uses $\mathsf{set}(f, h, x_i^{\mathrm{D}}, v) \Rightarrow h'$ for updating frame f in heap h by assigning v to slot x_i^{D} to produce a new frame h'. Setting a slot fails if the slot has not been allocated. Note that Rule [EvAsgn] defines assignment not just for record fields but also for variables, which makes variables bound by functions mutable. Our framework currently does not distinguish between mutable and immutable variables, but we expect it would be straightforward to introduce such a distinction, e.g. by classifying declarations into separate namespaces.

4.3 Type Soundness

The rules in Figure 21 summarize the value typing judgment for L2. Here, null-pointer exceptions are viewed as well-typed values of any kind, which allows such exceptions to occur anywhere in well-typed programs. Theorem 3 states type preservation for L2.

▶ **Theorem 3** (Type preservation for L2). *Suppose* $\mathsf{goodHeap}(h_1)$ *and* $\mathcal{S}_{h_1}(f) = s$, *as well as* $\overset{\mathrm{B}}{\vdash} e^s$ *and* $\overset{\mathrm{T}}{\vdash} e^t$ *hold. Then, for any* v *and* h_2, $f \vdash e^{s,t}/h_1 \Rightarrow v/h_2$ *implies* $\mathsf{goodHeap}(h_2)$ *and* $h_2 \overset{\mathrm{V}}{\vdash} v : t$.

As argued in connection with the proof of Theorem 2 in Section 3.6, we can extend this proof to a proof of type soundness by adding cases for "going wrong". The wrong cases, which are given in the accompanying technical report [17] follow along the same lines as the cases in the preservation proof.

4.4 From Records to Classes

Our accompanying technical report [17] describes L3, an extension of L2 which replaces records by classes with inheritance, sub-typing, and overriding. Inheritance and sub-typing are modeled by representing run-time objects as a hierarchy of frames, one frame for each super-class in the hierarchy. Paths to object fields may contain edges that traverse this hierarchy. This poses interesting challenges for looking up fields at run time.

Figure 22 provides a small but illustrative example L3 program, its scope graph, and its object frame structure. Consider the expression: `(fun (a : A) { a.x })(new B)`. The path of the field access expression `a.x` is calculated based on `a` being an object of type `A`; i.e., $\mathbf{E}(\mathbf{I}, \mathtt{s2}) \cdot \mathbf{D}(\mathtt{x})$, where `s2` is the scope for the class `A`. But applying this path to the actual field of the value gets us to a subtype of `A`, namely the scope of the `B` object, which only provides an `x` slot via its import edge! The actual path to the desired `x` slot is thus: $\mathbf{E}(\mathbf{I}, \mathtt{s3}) \cdot \mathbf{E}(\mathbf{I}, \mathtt{s2}) \cdot \mathbf{D}(\mathtt{x})$. In order to deal with this mismatch between the statically computed path and the actual path, we *upcast* the `B` frame (instantiating scope `s3`) bound to `a` before attempting to access fields through it. Upcasting follows a chain of import edges until it arrives at a frame that instantiates the statically expected type, in this case the `A` frame

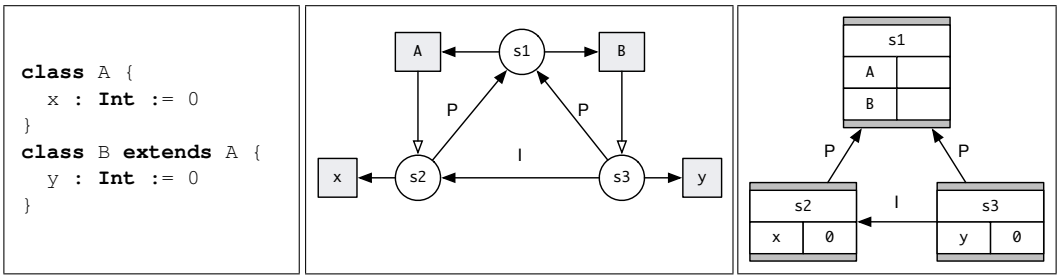


Figure 22 An example program, its scope graph, and its object frame structure.

(instantiating scope `s2`). The technical report [17] and Coq development provide the full details.

5 Garbage Collection

So far, we have described how the heap grows by adding frames. Any realistic language implementation also needs to consider how unused frames can be reclaimed. To this end, we now give a high-level specification of sound reachability-based garbage collection of frames in our setting, and verify that several standard GC strategies refine this specification. We leave actual implementation of concrete collectors as future work.

We model garbage collection as removal of frames from the heap map; the removed frame identifiers then become fresh again for subsequent frame allocations. It is safe to remove a frame exactly when doing so would not change subsequent observable program behavior [12]. Since this is an undecidable criterion, collectors instead use an approximation based on whether there are live pointers to the frame; if not, the frame certainly cannot be accessed again (assuming pointers are unforgeable) and hence may be safely removed.

A frame may be referenced in one of two ways, either (i) from a *root* pointer in the execution state of the program, or (ii) from another frame. We focus first on category (ii), for which we can give a simple and largely language-independent characterization. All our formal definitions are in Figure 23.

Frame-to-frame pointers. We write $h \vdash f \rightsquigarrow f'$ if frame f in heap h makes a *direct reference* to frame f', either through a link or via a slot value v. The latter case, which we write $v \rightsquigarrow f'$, depends on the definition of values in our language. A frame f' is *reachable* from another frame f if there is a sequence of direct references from f to f'; this is just the reflexive transitive closure of the reference relation, so we write it $h \vdash f \rightsquigarrow^* f'$. We write $h \vdash \not\rightsquigarrow f$ if no frame in h references f.

The key observation is that it is safe to remove any set of frames fs from a heap h provided that no frame in fs is referenced from the *resulting* heap h', written $\mathsf{safeRemoval}(h, fs, h')$. In other words, a safe removal is one that does not produce any new dangling pointers. We then have the following easy lemma:

▶ **Lemma 4** (Safe removal preserves heap invariants)**.** *Suppose* $\mathsf{goodHeap}(h)$ *holds. Then, for any set of frames* fs *and heap* h'*,* $\mathsf{safeRemoval}(h, fs, h')$ *implies* $\mathsf{goodHeap}(h')$*.*

Roots. How to track roots into the heap from the program state depends on the details of the language and the semantic approach. For the big-step semantic style used in this paper, most live frames are reachable from the "current" frame (f in the judgments $f \vdash a/h \Rightarrow v/h'$).

References from values

$\mathsf{ClosV}(x_i^{\mathrm{D}}, a, f) \rightsquigarrow f$ [RefClos]

$\mathsf{RecordV}(f) \rightsquigarrow f$ [RefRecord]

References from frames

$$\frac{\mathsf{get}(f, h, x_i^{\mathrm{D}}) \Rightarrow v \quad v \rightsquigarrow f'}{h \vdash f \rightsquigarrow f'}$$ [RefSlot]

$$\frac{\mathcal{K}_h(f)(l)(s) = f'}{h \vdash f \rightsquigarrow f'}$$ [RefLink]

Unreferenced frames $h \vdash \not\rightsquigarrow f \triangleq \forall f' \in Dom(h).\ h \vdash f' \not\rightsquigarrow f$

Removing frames from heaps

$\mathsf{safeRemoval}(h, fs, h') \triangleq h' = (h - fs) \wedge \forall f \in fs.\ h' \vdash \not\rightsquigarrow f$

$\mathsf{removeUnreferenced}(h, f, h') \triangleq h' = (h - \{f\}) \wedge h \vdash \not\rightsquigarrow f$

$\mathsf{removeAllUnreachable}(h, rs, fs, h') \triangleq h' = (h - fs) \wedge fs = \{f \mid \forall f' \in rs.\ h \vdash f' \not\rightsquigarrow^* f\}$

where $(h - fs)$ is the result of removing all frames in fs from h

Expression evaluation (selected rules) $f, rs \vdash a/h \Rightarrow v/h'$

$$\frac{f, rs \vdash a_1/h_1 \Rightarrow \mathsf{ClosV}(x_i^{\mathrm{D}}, a', f')/h_2 \quad f, \{f'\} \cup rs \vdash a_2/h_2 \Rightarrow v/h_3 \quad \mathsf{initFrame}(\mathcal{S}(a'), \{\mathbf{P} \mapsto \{\mathcal{S}_{h_3}(f') \mapsto f'\}\}, \{x_i^{\mathrm{D}} \mapsto v\})/h_3 \Rightarrow f''/h_4 \quad f'', \{f\} \cup rs \vdash a'/h_4 \Rightarrow v'/h_5}{f, rs \vdash a_1(a_2)/h_1 \Rightarrow v'/h_5}$$ [EvApp]

$$\frac{\mathsf{safeRemoval}(h, fs, h') \quad fs \cap (\{f\} \cup rs) = \emptyset \quad f, rs \vdash a/h' \Rightarrow v/h''}{f, rs \vdash a/h \Rightarrow v/h''}$$ [EvGC]

Figure 23 Definitions for Garbage Collection.

However, evaluation of certain big-step rules may introduce other frames, not reachable from the current frame, that must be temporarily registered as roots. To handle these, we add an additional root frame set component rs to the configurations described by the rules. Figure 23 shows the revised rule for application to illustrate how the auxiliary root set is used, here in two different ways: to save the closure frame while evaluating the argument and to save the calling context frame while evaluating the function body.

Several other rules also augment the root set; the need for auxiliary roots is unfortunately a bit ad-hoc and language-specific. We could limit the number of auxiliary roots by requiring source programs to be in A-normal form or continuation-passing style, which both have the effect of putting more rule-internal values into named variables, and hence into the current frame.

Type-safe collection. Finally, we can add Rule [EvGC] for garbage collection shown in Figure 23, which can be applied non-deterministically prior to any of the syntax-directed rules. The resulting system still enjoys type soundness. The proof is straightforward once we strengthen the inductive invariant at each big step to say that $\{f\} \cup rs \subseteq Dom(h)$. The proof case for Rule [EvGC] relies on Lemma 4.

Refinements. To show that safeRemoval is a reasonable specification, we note that it can be refined to specifications of two well-known GC algorithms. removeUnreferenced specifies removal of a single unreferenced frame; it models one step in a reference counting collection. removeAllUnreachable specifies removal of *all* frames unreachable from an arbitrary root set; it models a *complete* tracing-based collector, such as a mark-and-sweep or copying collector.

▶ **Lemma 5** (Unreferenced and unreachable frames are safe to remove)**.** *It holds that:*
(a) removeUnreferenced(h, f, h') *implies* safeRemoval$(h, \{f\}, h')$*; and*
(b) removeAllUnreachable(h, rs, fs, h') *implies* safeRemoval(h, fs, h')*.*

We have not yet developed a concrete GC implementation, but writing one should be straightforward. The heap-traversal part of an implementation can be language-independent except for determining references from values. Note that, for many languages, an implementation can use the scope labels on frames to detect all references efficiently, without the need for tagging individual pointers. This follows from the goodFrame property, which states that every frame slot corresponds to a statically known typed declaration in the scope; if the language is sufficiently monomorphic or restricts polymorphism to boxed types, we can consult the scope description to determine whether or not the slot contains a pointer.

6 Discussion

This paper presents a systematic correspondence between static name binding and binding at run time. We have also proposed the correspondence as a guiding principle for dynamic semantic specification, and a language-independent principle for proving type soundness. In this section we discuss our approach and compare with previous work on type soundness and, more generally, on representing and reasoning about semantics and memory layout.

Type soundness. Milner [11] summarized the essence of type soundness in his famous catch-phrase: "well-typed programs cannot go wrong". The common point of type soundness proofs is that they provide this guarantee. Other than that, type soundness proofs come in many different varieties and flavours, depending on both the underlying semantic style (such as big-step vs. small-step) and the underlying language being specified.

Semantic specification and type soundness. Wright and Felleisen's *syntactic approach to type soundness* [28] argues in favor of using small-step reduction semantics and evaluation contexts for type soundness proofs. This avoids some of the shortcomings of big-step semantics, namely that big-step rules do not distinguish getting stuck and diverging. The small-step style is also better suited for formalizing semantics with concurrency and/or interleaving. However, this kind of specification does not correspond to how programming language interpreters are typically implemented.

This paper uses big-step semantics (or *natural semantics* [7]) for type soundness. In order to prove type soundness we have been following in the footsteps of Milner [11] and added explicit rules for going wrong to our language. An inherent danger of this approach is that leaving out such a wrong rule may render the type soundness theorem vacuous. We stress that the approach of using scopes to describe frames is by no means limited to big-step semantics: we expect it to be equally applicable to other styles of semantic specification, including small-step SOS [16], reduction semantics [5], definitional interpreters [18], and abstract machines. We chose big-step semantics because it corresponds to how programming language interpreters are typically implemented. In the future, we plan to use I-MSOS [13] and DynSem [26] to make our big-step rules even more concise. We also plan to investigate ways of alleviating some of the drawbacks of big-step type soundness proofs proposed in the literature, such as:

- using coinductive big-step semantics to represent diverging computations and proving big-step progress, following Leroy and Grall [10];

- checking that wrong rules are correctly defined either by using a *coverage lemma* as proposed by Ernst et al. [4] or using Charguéraud's *pretty-big-step* style [1] with a novel big-step progress predicate, which subsumes explicit wrong rules in a safe way (e.g., failure to add sufficient rules for progress makes it impossible to prove type soundness);
- using a functional definitional interpreter instrumented with a clock (or "fuel") which counts down with each function call [22, 15, 19].

The modularity of the small-step approach in [28] comes from the use of reduction semantics, which supports localizing new patterns of name binding to particular constructs, such that proofs of existing constructs do not need to change. In contrast, our approach using scopes to describe frames scales to deal with name binding of both functional and imperative features in a uniform manner. Thus, there is no need to use different stores or notions of substitution or to localize these to particular constructs: name binding is uniformly handled both in the dynamic semantics and in the proof.

A selling point of the syntactic approach is that it scales to prove the type soundness of an ML-like language with Hindley-Milner polymorphism such that the structure of proofs change relatively little as new features are added. Indeed, many of the challenges with substitution in [28] stem from dealing with polymorphism and references. We believe that describing scopes as frames will scale to deal with polymorphic type inference, and suspect it might alleviate some of the difficulties inherent to substitution lemmas, but leave it to future work to verify this.

Default values. A shortcoming of our approach compared with, e.g., Wright and Felleisen [28], is our use of default values. Default values make the proofs easy, but introduce the risk of dereferencing null values. In order to deal with function types, we proposed to generate "default functions" too, but this approach is restricted to simple type systems, since the problem of finding inhabitants of a given type quickly becomes undecidable. In the future, we plan to investigate extending our type soundness principle to temporarily allow a frame to be ill-typed, so long as it is initialized when it is accessed. Nipkow et al.'s [9] formalization of Java's definite assignment analysis could provide a useful guide to this end.

Modeling memory in programming language semantics. In addition to the different semantic specification styles, there is a proliferation of ways to deal with name binding and memory management in semantic specifications. For example, consider a type soundness proof for Java-like languages, where formalization of name binding and memory ranges from simple states mapping identifiers and references to values, as used by Drossopoulou and Eisenbach [3], to untyped frames [24] relying on traditional environments for typing, to use of ad-hoc lookup functions (or visibility predicates) uniquely defined to resolve a specific kind of identifiers (e.g. classes or fields) as used in Featherweight Java [6] or Jinja [9].

Similarly, considering the state of affairs in semantic specification frameworks gives a similarly muddled picture. These frameworks can be roughly categorized into two camps: those that deal with binding via substitution (redex [8], Ott [21]), and those that provide support for ad hoc binding via auxiliary entities (K [20], funcons [2]).

None of these pre-existing approaches hits the same sweet spot as our framework: it corresponds to how memory is often organized in real implementations (as also remarked by Syme [24, page 7]); it scales to deal with various models of name binding and memory in a uniform manner; and it provides a language-independent principle for proving the absence of binding and typing errors.

We also remark that our compartmentalization of binding and typing as separate concerns and separate sets of rules in theory allows us to prove the absence of binding errors and typing errors separately. This may be useful for dynamically-typed languages, where proving the absence of typing errors might not be a concern, but the absence of binding errors is.

7 Conclusion

This paper presented a systematic and uniform correspondence between static scopes and dynamic frames. This correspondence provides a mostly separate specification of binding and typing. This is a novelty compared to traditional approaches, where typing rules usually describe both name binding and typing, occasionally relying on ad hoc predicates (such as those found in, e.g., Featherweight Java). The scopes-as-frames paradigm supports uniform and straightforward type soundness proofs for a number of different language features, as demonstrated by applying it to a simple functional language (Section 3) and a language with first-class functions and records (Section 4). Section 5 shows how the invariants also support verifying the safety of a number of different garbage collection schemes. Section 4.4 discusses, and our accompanying tech report [17] shows, how the approach also scales to a class-based object-oriented language with sub-typing and run-time upcasting.

In future work, we plan to investigate how the approach scales to larger languages including languages with type-level polymorphism; generating Coq infrastructure for mechanizing type soundness proofs using the language-independent framework for well-boundness and well-typedness developed for this paper; and how to derive efficiently executable prototype interpreters based on dynamic semantic specifications in DynSem.

Acknowledgments. We thank Sebastian Erdweg, Peter D. Mosses, and the anonymous reviewers for their feedback on previous versions of this paper.

References

1 Arthur Charguéraud. Pretty-big-step semantics. In Matthias Felleisen and Philippa Gardner, editors, *ESOP'13*, volume 7792 of *LNCS*, pages 41–60. Springer, 2013. doi:10.1007/978-3-642-37036-6_3.

2 Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. Reusable components of semantic specifications. *Transactions on Aspect-Oriented Software Development*, 12:132–179, 2015. doi:10.1007/978-3-662-46734-3_4.

3 Sophia Drossopoulou and Susan Eisenbach. Java is type safe – probably. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97*, volume 1241 of *LNCS*, pages 389–418. Springer, 1997.

4 Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL'06*, pages 270–282. ACM, 2006. doi:10.1145/1111037.1111062.

5 Matthias Felleisen. *The Calculi of λ-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-order Programming Languages*. PhD thesis, Indiana University, 1987.

6 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

7 Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS'87*, volume 247 of *LNCS*, pages 22–39. Springer, 1987.

8 Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: on the effectiveness of lightweight mechanization. In John Field and Michael Hicks, editors, *POPL'12*, pages 285–296. ACM, 2012. doi:10.1145/2103656.2103691.

9 Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *TOPLAS*, 28(4):619–695, 2006. doi:10.1145/1146811.

10 Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009. doi:10.1016/j.ic.2007.12.004.

11 Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

12 J. Gregory Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *FPCA'95*, pages 66–77, 1995.

13 Peter D. Mosses and Mark J. New. Implicit propagation in structural operational semantics. *ENTCS*, 229(4):49–66, 2009. doi:10.1016/j.entcs.2009.07.073.

14 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *ESOP'15*, volume 9032 of *LNCS*, pages 205–231. Springer, 2015.

15 Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In *ESOP'16*, volume 9632 of *LNCS*, pages 589–615. Springer, 2016. doi:10.1007/978-3-662-49498-1_23.

16 Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

17 Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. Scopes describe frames: A uniform model for memory layout in dynamic semantics. Technical Report TUD-SERG-2016-010, Delft University of Technology, Programming Languages Research Group, Delft, The Netherlands, 2016.

18 John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

19 Tiark Rompf and Nada Amin. From F to DOT: Type soundness proofs with definitional interpreters, 2015. URL: http://arxiv.org/abs/1510.05216.

20 Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. doi:10.1016/j.jlap.2010.03.012.

21 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010. doi:10.1017/S0956796809990293.

22 Jeremy G. Siek. Type safety in three easy lemmas, May 2013. URL: http://siek.blogspot.co.uk/2013/05/type-safety-in-three-easy-lemmas.html.

23 Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):11–49, 2000.

24 Don Syme. Proving Java type soundness. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 83–118. Springer, 1999.

25 Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In Martin Erwig and Tiark Rompf, editors, *PEPM'16*, pages 49–60. ACM, 2016. doi:10.1145/2847538.2847543.

26 Vlad A. Vergu, Pierre Néron, and Eelco Visser. DynSem: A DSL for dynamic semantics specification. In Maribel Fernández, editor, *RTA'15*, volume 36 of *LIPIcs*, pages 365–378.

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.RTA.2015.365.

27 Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël D. P. Konat. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Onward!'14*, pages 95–111. ACM, 2014. doi:10.1145/2661136.2661149.

28 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994.

Lightweight Session Programming in Scala*

Alceste Scalas[1] and Nobuko Yoshida[2]

1 **Imperial College London, UK**
`a.scalas@imperial.ac.uk`
2 **Imperial College London, UK**
`n.yoshida@imperial.ac.uk`

Abstract

Designing, developing and maintaining concurrent applications is an error-prone and time-consuming task; most difficulties arise because compilers are usually unable to check whether the inputs/outputs performed by a program at runtime will adhere to a given protocol specification.

To address this problem, we propose *lightweight session programming in Scala*: we leverage the native features of the Scala type system and standard library, to introduce (1) a representation of *session types* as Scala types, and (2) a library, called `lchannels`, with a convenient API for session-based programming, supporting *local* and *distributed* communication. We generalise the idea of *Continuation-Passing Style Protocols (CPSPs)*, studying their formal relationship with session types. We illustrate how session programming can be carried over in Scala: how to formalise a communication protocol, and represent it using Scala classes and `lchannels`, letting the compiler help spotting protocol violations. We attest the practicality of our approach with a complex use case, and evaluate the performance of `lchannels` with a series of benchmarks.

1998 ACM Subject Classification D.1.3 Concurrent Programming, D.3.1 Formal Definitions and Theory, F.3.3 Studies of Program Constructs – Type structure

Keywords and phrases session types, Scala, concurrency

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.21

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.2.1.11`

1 Introduction and motivation

Concurrent and distributed applications are notoriously difficult to design, develop and maintain. One of the main challenges lies in ensuring that software components interact according to some predetermined *communication protocols* describing all the valid message exchanges. Such a challenge is typically tackled at *runtime*, e.g. via testing and message monitoring.

Unfortunately, depending on the number of software components and the complexity of their protocols, tests and monitoring routines can be costly to develop and to maintain, as software and protocols evolve.

Consider the message sequence chart in Figure 1: it is based on an example of "actor protocol" from [26] (slide 42), and schematises the authentication procedure of an application server. A client connects to a frontend, trying to retrieve an active session by its Id; the

* Work partly supported by: EPSRC EP/K011715/1, EP/K034413/1 and EP/L00058X/1, and EU project FP7-612985 *UpScale*.

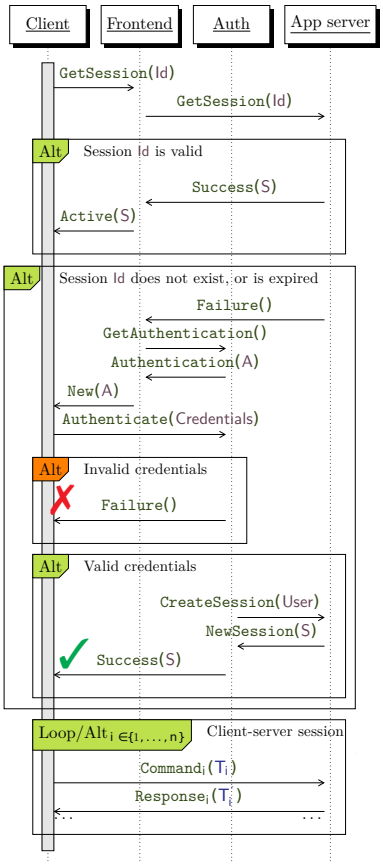


Figure 1 Server with frontend.

frontend queries the application server: if Id is valid, the client gets an `Active(S)` message with a session handle S, which can be used to perform the command/response loop at the bottom; otherwise, the client must authenticate: the frontend obtains an handle A from an authentication server, and forwards it to the client with a `New(A)` message. The client must now use A to send its credentials (through an `Authenticate` message); if they are *not* valid, the authentication server replies `Failure()`; otherwise, it retrieves a session handle S and sends `Success(S)` to the client, who uses S for the session loop (as above). In this example, four components interact with intertwined protocols. Ensuring that messages are sent with the right type and order, and that each component correctly handles all possible responses, can be an elusive and time-consuming task. Runtime monitoring/testing can detect the presence of communication errors, but cannot guarantee their absence; moreover, protocols and code may change during the life cycle of an application – and monitoring/testing procedures will need to be updated. Compile-time checks would allow to reduce this burden, lowering software maintenance costs.

CPS protocols in Scala. The developers of the Akka framework [28] have been addressing these challenges, in the setting of actor-based applications. Standard actors communicate in an *untyped* way: they can send each other *any* message, *anytime*, and must check at runtime whether a given protocol is respected. Akka developers are thus trying to leverage the Scala type system to obtain *static* protocol definitions and *compile-time* guarantees on the absence

```
case class GetSession(id: Int,
                      replyTo: ActorRef[GetSessionResult])

sealed abstract class GetSessionResult
case class New(authc: ActorRef[Authenticate])
    extends GetSessionResult
case class Active(service: ActorRef[Command])
    extends GetSessionResult

case class Authenticate(username: String, password: String,
                        replyTo: ActorRef[AuthenticateResult])

sealed abstract class AuthenticateResult
case class Success(service: ActorRef[Command])
      extends AuthenticateResult
case class Failure()  extends AuthenticateResult

sealed abstract class Command
// ... case classes for the client-server session loop ...
```

Figure 2 Akka Typed: protocol of client in Fig. 1.

```
1 def client(frontend: ActorRef[GetSession]) = {
2   val cont = spawn[GetSessionResult] {
3     case New(a) => doAuthentication(a)
4     case Active(s) => doSessionLoop(s)
5   }
6   frontend ! GetSession(42, cont)
7 }
```

Figure 3 Actor spawning (pseudo code).

of communication errors. Their tentative solution has two parts. The first is Akka Typed [29]: an experimental library with actors that can only receive messages via references of type `ActorRef[A]`, which in turn only allow to send `A`-typed messages. The second is what we dub *Continuation-Passing Style Protocols (CPSPs)*: sets of message classes that represent sequencing with a `replyTo` field, of type `ActorRef[B]`. By convention, `replyTo` tells where the message recipient should send its `B`-typed answer: Fig. 2 (based on [26], slide 41) shows the CPSPs of the client in Fig. 1.

In practice, a `replyTo` field can be instantiated by *producing a "continuation actor"* that handles the next step of the protocol. Fig. 3 shows a client that, *before* sending `GetSession` to the frontend (line 6), *spawns a new actor* accepting `GetSessionResult` messages. Then, `cont` (line 2) has type `ActorRef[GetSessionResult]`, and is sent as `replyTo`: the frontend should send its `New`/`Active` answer there. This creates a conversation between the client and frontend: the message sender produces a "continuation", and the receiver should use it.

Opportunities and limitations. CPSPs have the appealing feature of being *standard* Scala types, checked by its compiler, and giving rise to a form of structured interaction in Akka. However, their incarnation seen above has some shortcomings. First and foremost, they are a rather low-level representation, not connected with any established, high-level formalisation of protocols and structured interaction. Hence, non-trivial protocols with branching and recursion (e.g. the one in Fig. 1) can be hard to write and understand in CPS; even message ownership and sequencing may be non-obvious: e.g., determining who sends `Failure` in Fig. 2, and whether it comes before or after another message, can take some time. Moreover, the CPSPs in Fig. 2 seems to imply that some continuations should be used *exactly once* – but this intuition is not made explicit in the types. E.g., in Fig. 3, `frontend` and `cont` are both `ActorRef`s – but the actor referred by `frontend` might accept *multiple* `GetSession`

requests, whereas the one referred by `cont` (spawned on lines 2–5) might just wait for *one* `New`/`Active` message, spawn another continuation actor, and terminate. Arguably, the type of `cont` should convey whether sending more than one message is an error.

Our contribution: lightweight session programming in Scala. We address the challenges and limitations above by proposing *lightweight session programming in Scala* – where "lightweight" means that our proposal does *not* depend on language extensions, nor external tools, nor specific message transport frameworks. We generalise the idea of CPSP, relating it to a well established formalism for the static verification of concurrent programs: *session types* [19, 20, 39]. We present a library, called `lchannels`, offering a simplified API for session programming with CPSPs, supporting network-transparent communication. Albeit the Scala type checker does not cater for all the static guarantees provided by session-typed languages (mostly due to the lack of *static linearity checks*), we show that `lchannels` and CPSPs allow to represent protocol specifications as Scala types, and write session-based programs in a rather natural way, guaranteeing *protocol safety*: i.e., once a session starts, no out-of-protocol messages can be sent, and all valid incoming messages are handled. We show that typical protocol errors are detected at compile-time – except for *linearity errors*: `lchannels` checks them at runtime, reminding the typical usage of Scala `Promise`s/`Future`s.

This work focuses on Scala since we leverage several convenient features of the language and its standard library: object orientation, parametric polymorphism with declaration-site variance, first-class functions, labelled union types (`case class`es), `Promise`s/`Future`s; yet, our approach could be adapted (at least in part) to any language with similar features.

Outline of the paper. In § 2, we summarise session types, explaining the difficulties in their integration in a language like Scala, and how we overcome them by exploiting an encoding into *linear types for I/O*. In § 3 we introduce `lchannels`, a library for type-safe communication over asynchronous *linear* channels. In § 4 we explain, via several examples, how session programming can be carried over in Scala, by using `lchannels` and representing session types as CPSPs, according to a *session-based software development approach* (§ 4.2). § 5 presents optimisations and extensions of `lchannels`, achieving message transport abstraction and network-transparent communication. In § 6 we show the practicality of our approach by implementing the case study in Fig. 1, and evaluating the performance of `lchannels` – particularly, its message delivery speed w.r.t. other inter-process communication methods. In § 7 we give a formal foundation to § 4, proving crucial results about *duality*/*subtyping* of session types represented in Scala, and overcoming technical difficulties in the transition from a structural to nominal types (e.g., different handling of recursion). We discuss related works in § 8, and conclude in § 9 – showing how our approach can be adapted to other communication frameworks.

Online resources. Due to space limits, we include proofs, benchmarking details and other materials in `http://www.doc.ic.ac.uk/research/technicalreports/2015/#7`. For the latest version of `lchannels`, visit `http://alcestes.github.io/lchannels/`.

2 Programming with session types: background and challenges

We now summarise the features of languages based on *binary session types* (§ 2.1) and their notions of *duality* and *subtyping* (§ 2.2). We then explain their relationship with *linear I/O types* (§ 2.3), and give an overview of our strategy for representing them in Scala (§ 2.4).

2.1 Background: binary session types in a nutshell

Session types regulate the interaction of *processes* communicating through *channels*; each channel has two *endpoints*, and the intuitive semantics is that all values sent on one endpoint can be received on the other *in the same order* – a bidirectional FIFO model akin e.g. to TCP/IP sockets. A session type says how a process is expected to use a channel endpoint. Let $\mathbb{B} = \{\mathsf{Int}, \mathsf{Bool}, \mathsf{Unit}, \ldots\}$ be a set of *basic types*. A session type S has the following syntax:

$$S \;::=\; \&_{i\in I}?\mathtt{l}_i(T_i).S_i \;\mid\; \oplus_{i\in I}!\mathtt{l}_i(T_i).S_i \;\mid\; \mu_X.S \;\mid\; X \;\mid\; \mathbf{end} \qquad T \;::=\; \mathbb{B} \mid S \text{ (closed)}$$

where $I \neq \varnothing$, recursion is guarded, and all $\mathtt{l}_i$ range over pairwise distinct *labels*. T denotes a *payload type*. The *branching type* (or *external choice*) $\&_{i\in I}?\mathtt{l}_i(T_i).S_i$ requires the process to receive one input of the form $\mathtt{l}_i(T_i)$, for any $i \in I$ chosen at the *other* endpoint; then, the channel must be used according to the *continuation type* S_i. The *selection type* (or *internal choice*) $\oplus_{i\in I}!\mathtt{l}_i(T_i).S_i$, instead, requires the program to choose and perform one output $\mathtt{l}_i(T_i)$, for some $i \in I$, and continue using the channel according to S_i. $\mu_X.S$ is a *recursive* session type, where μ binds X, and X is a *recursion variable*. We say that S *is closed* iff all its recursion variables are bound. **end** is a *terminated* session with no further inputs/outputs. Note that a payload type T can be either a basic or a session type: hence, channel endpoints allow to send/receive e.g. integers, strings, or other channel endpoints.

▶ Remark 2.1. We use $\oplus/\&$ as infix operators, omitting them in singleton choices. We often omit **end** and Unit: $?\mathtt{A}\big(!\mathtt{B}(\mathsf{Int}) \oplus !\mathtt{C}\big)$ stands for $\&\{\,?\mathtt{A}(\mathsf{Unit}).\oplus\{!\mathtt{B}(\mathsf{Int}).\mathbf{end}\,,\,!\mathtt{C}(\mathsf{Unit}).\mathbf{end}\}\,\}$.

For example, the type S_h below describes the client endpoint of a "greeting protocol":

$$S_\mathrm{h} \;=\; \mu_X.\Big(!\mathtt{Greet}(\mathsf{String}).\big(?\mathtt{Hello}(\mathsf{String}).X \;\&\; ?\mathtt{Bye}(\mathsf{String}).\mathbf{end}\big) \;\oplus\; !\mathtt{Quit}.\mathbf{end}\Big)$$

The client can send either `Quit` and **end** the session, or `Greet(String)`; in the second case, it might receive from the server either `Bye(String)` (**end**ing the session), or `Hello(String)`: in the second case, the session continues recursively.

Programming languages that support session types are usually based on session-π – i.e., a version of π-calculus [31] extended with session operators. A client respecting S_h would be implemented as `hello(c)` in Fig. 4 (left): `c` is a S_h-typed channel endpoint, `!` is a language primitive for selecting and sending messages, and `?` for branching (i.e., receiving and pattern matching messages). The type system ensures that `c` is used according to S_h, guaranteeing:

- **S1.** *safety:* no out-of-protocol I/O actions are allowed. E.g., `c` can initially be used *only* to send `Greet`/`Quit` (lines 3/8), no outputs are allowed when S_h expects `c` to receive (line 4), no inputs when S_h expects `c` to send (lines 3,8), no I/O when S_h has **end**ed (line 6);
- **S2.** *exhaustiveness:* when receiving a message, all outcomes allowed by the type must be covered. E.g., the client must handle *both* `Hello` and `Bye` answers (lines 4–6);
- **S3.** *output linearity:* if S_h prescribes an output, it must occur *exactly once*. E.g., after receiving `Hello`, the client *must* send `Greet` or `Quit` (as in the recursive call of line 5);
- **S4.** *input linearity:* similarly, if S_h prescribes an input, it must occur *exactly once*. E.g., after sending `Greet`, the client *must* receive the response (as in line 4).

2.2 Background: safe, deadlock-free interaction via duality/subtyping

A session-typed language ensures correct run-time interaction by statically checking that the two endpoints of a channel are used *dually*. The *dual of* S, written $\overline{S}$, is defined as:

$$\overline{\&_{i\in I}?\mathtt{l}_i(T_i).S_i} = \oplus_{i\in I}!\mathtt{l}_i(T_i).\overline{S_i} \qquad \overline{\oplus_{i\in I}!\mathtt{l}_i(T_i).S_i} = \&_{i\in I}?\mathtt{l}_i(T_i).\overline{S_i}$$
$$\overline{\mu_X.S} = \mu_X.\overline{S} \qquad \overline{X} = X \qquad \overline{\mathbf{end}} = \mathbf{end}$$

```
1 def hello(c: S_h): Unit = {
2   if (...) {
3     c ! Greet("Alice")
4     c ? {
5       case Hello(name) => hello(c)
6       case Bye(name) => ()
7     }
8   } else { c ! Quit() }
9 }
```

```
1 def lHello(c: LinOutChannel[?]): Unit = {
2   if (...) {
3     val (c2in, c2out) = createLinChannels[?]()
4     c.send( Greet("Alice", c2out) )
5     c2in.receive match {
6       case Hello(name, c3out) => lHello(c3out)
7       case Bye(name) => ()
8     }
9   } else { c.send( Quit() ) }
10 }
```

Figure 4 Greeting protocol client (pseudo code): session types (left) vs. linear I/O types (right).

Intuitively, the internal/external choices of S are swapped in $\overline{S}$; hence, each client-side output is matched by a server-side input, and *vice versa*. In our example, `c` is a client-side endpoint that must be used according to S_{h}; the server-side dual channel endpoint has type:

$$\overline{S_{\text{h}}} = \mu X.\Big(?\texttt{Greet(String)}.\big(!\texttt{Hello(String)}.X \oplus !\texttt{Bye(String)}.\mathbf{end}\big) \;\&\; ?\texttt{Quit}.\mathbf{end}\Big)$$

Duality guarantees the *safe and deadlock-free interaction* of a client and server observing S_{h} and $\overline{S_{\text{h}}}$: no unexpected messages are sent/received, and the session *progresses* until its **end**.

Such a guarantee is made more flexible via *session subtyping* [13]. Consider the type $S_{\text{h2}} = !\texttt{Quit}$, and its implementation on the right: since `hello2` only outputs `Quit` on `c2`, it would also behave safely on a S_{h}-typed channel endpoint `c`. In fact, in a session-typed language we have $S_{\text{h}} \leqslant S_{\text{h2}}$– i.e., an S_{h}-typed channel endpoint can always be used in place of an S_{h2}-typed one; hence, invoking `hello2(c)` is allowed – and such a client program would interact safely and without deadlocks with a server observing $\overline{S_{\text{h}}}$.

```
1 def hello2(c2: S_h2): Unit = {
2   c2 ! Quit()
3 }
```

2.3 From session-typed to linearly-typed programs

Unfortunately, integrating session types into a "mainstream" programming language is not trivial: they require sophisticated type system features. *Safety/exhaustiveness* can be achieved by letting `c`'s type evolve according to S_{h} after each I/O action – but most type systems assign a fixed type to each variable; *I/O linearity* checks require linearity analysis; *internal/external choices*, *session subtyping* and *duality* need dedicated type-level machinery.

In this paper, we show how *session programming* can be carried over in Scala, recovering part of the static guarantees provided by session types. We take inspiration from the encoding of session-π into *standard π-calculus with variants and linear I/O types* [8]: the key idea is that session-π and session types can be encoded in a more basic language and type system that do *not* natively support session primitives (e.g., internal/external choices and duality), by adopting a "continuation-passing style" interaction over *linear* input/output channel endpoints that are used *exactly once*. In particular, [8] (Theorems 1, 2) proves that a process using variants, linear I/O types and CPS interaction can precisely mirror the typing and the runtime communications of a session typed process.

An intuition of our approach is given in Fig. 4 (right), where `lHello` is the "linearly encoded" version of `hello`. Its argument `c` is a *linear output channel endpoint* that carries a *single* value (whose type is left unspecified, for now). On line 3, it creates a new pair of *linear channels endpoints*, which can carry another single value of some (again unspecified) type: intuitively, what is sent on `c2out` becomes available in `c2in`. On line 4, `c` is used to send a `Greet` message – which *also carries* `c2out`. Then, the recipient of `Greet` and `c2out` is expected to use the latter to continue the session – i.e., send either `Hello` or `Bye`. On line 5, `c2in` is used to receive such an answer, and the result is matched against `Hello` and `Bye`; the

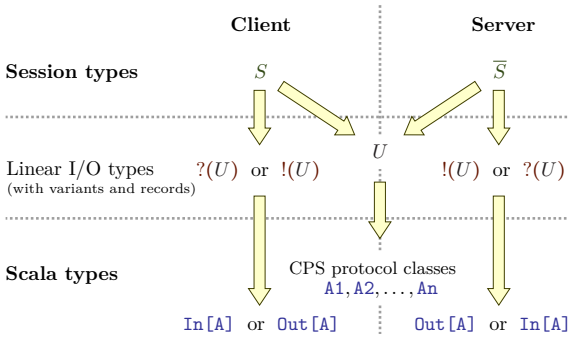


Figure 5 From session types to Scala types.

latter carries no continuation channel, i.e. the session has ended (line 7); the former, instead, carries a linear (output) channel endpoint `c3out`, that is used to continue the session with a recursive call (line 6). Note that all channel endpoints received/created in `lHello` are either used exactly once (`c`, `c2in`, `c3out`), or sent to some other process (`c2out`).

A crucial difference between `hello` and `lHello` is that in the latter, *each variable has a constant type*. This suggests that, although the Scala type checker cannot check linearity, it *might* be leveraged to obtain a form of session typing, offering *safety* and *exhaustiveness* for programs written in "linear CPS", like `lHello`. Then, as seen in § 2.2, we could also obtain *safe and deadlock-free interaction* – provided that a program creates, uses or sends its linear channel endpoints according to [8], and the other program involved in a session interacts in a "dual" way. However, the pseudo-code of Fig. 4 (right) highlights four Problems:

P1. we need to represent and implement *linear input and output channels*;
P2. we need to suitably *instantiate each* `?`*-type*, so to describe the same interactions of S_{h};
P3. we must *automate the creation, sending and use of linear channels*, offering an API that guides the CPS interactions prescribed in [8], and allows to write code similar to `hello`;
P4. we need to handle *session subtyping and duality* in the Scala type system.

2.4 From session types to session programming in Scala: an outline

In the rest of the paper, we demonstrate how to tackle Problems **P1**–**P4**, staying close to the session/linear types theory, and yet achieving *practical* session programming in Scala. Our approach is summarised in Fig. 5. On top, we have a client and a server that should interact through a channel, whose protocol is described with dual session types S and $\overline{S}$. On the bottom, the same protocol is represented in Scala, as a set of *CPSP classes*, shared between the client and server, and similar to those discussed in § 1: they are used as parameters for `In[A]` and `Out[A]`, which implement respectively an input/output channel endpoint carrying a *single* value of type `A`. We extract such CPSP classes from S *or* $\overline{S}$, through an *encoding* represented by the arrows; such an encoding exploits an *intermediate generation of linear I/O types* (middle of Fig. 5), as detailed in § 7. We address **P1** in § 3, **P2** in § 4, **P3** in § 4.3, and **P4** in § 7.3.

3 `lchannels`, a (small) library for type-safe interaction

We now introduce `lchannels`, a Scala library providing *typed linear channels*. We designed the programmer interface to be close to the formal definition of linear channels (§ 7.2) –

```scala
abstract class In[+A] {
  def future: Future[A]
  def receive(implicit d: Duration): A = {
    Await.result[A](future, d)
  }
  def ?[B](f: A => B)(implicit d: Duration): B = {
    f(receive)
  }
}

abstract class Out[-A] {
  def promise[B <: A]: Promise[B]
  def send(msg: A): Unit    = promise.success(msg)
  def !(msg: A)                       = send(msg)
  def create[B](): (In[B], Out[B])
}
```

```scala
class LocalIn[+A](val future: Future[A]) extends In[A]

class LocalOut[-A](p: Promise[A]) extends Out[A] {
  override def promise[B <: A] = {
    p.asInstanceOf[Promise[B]] // Type-safe cast
  }
  override def create[B]() = LocalChannel.factory[B]()
}

object LocalChannel {
  def factory[A](): (LocalIn[A], LocalOut[A]) = {
    val promise = Promise[A]()
    val future = promise.future
    (new LocalIn[A](future), new LocalOut[A](promise))
  }
}
```

Figure 6 Linear channels in Scala: abstract classes (left) and local implementation (right).

notably, by reflecting their *co*/*contra*-variance. For simplicity, we shape the API and its basic implementation around `Promise`s/`Future`s from the Scala standard library [16], since they are familiar to Scala developers, and remarkably close to the expected usage of linear channel endpoints (§ 2.3): *(i)* a `Promise[A]` must be completed *exactly once* with an `A`-typed value `v`, and *(ii)* after completion, `v` becomes available on the corresponding `Future[A]`. Moreover, `Promise`s/`Future`s provide *asynchronous* message passing.

We present the `lchannels` API in § 3.1, and a simple implementation in § 3.2. We give further details, examples and extensions after showing the representation of session types as CPSP classes (§ 4) – which constitute the principal use case for `lchannels`.

3.1 The programmer interface

The cornerstones of `lchannels` are the abstract classes `Out[-A]` and `In[+A]`, representing channel endpoints allowing respectively to send and receive *one* `A`-typed value. Their slightly simplified declarations are shown in Fig. 6 (left).

The class `Out[-A]` is *contravariant* w.r.t. `A`[1]. Its `promise` (line 12) is expected to be eventually completed with the value to be sent; a crucial requirement is that `promise` *must be implemented as a constant*[2], to ensure that it will be completed only once. Note that due to the contravariance of `A`, the type of `promise` cannot be simply `Promise[A]`: the reason is that the latter is *invariant* w.r.t. `A`; the bounded type parameter `B <: A` allows to overcome this limitation. `send(msg)` and its alias `!` offer a simplified interface above `promise`, representing the selection/output operator of session-π (see Theorem 3.1). Finally, `Out`'s abstract method `create[B]()` returns a new pair of input/output channels carrying `B`: this method is used to create *continuation endpoints*, as seen in Fig. 4 (right, line 3).

The class `In[+A]` is *covariant* w.r.t. its type parameter `A`[3]. Its `future` will contain the value sent from the corresponding `Out` endpoint. The `receive` method offers a simplified interface over `future`: the implicit parameter `d` specifies how long to wait for an incoming message before raising a timeout error. The `?` method implements the typical *branching* operator of session-π: it takes a function `f: A => B`, and once a value `v` is `receive`d, it returns `f(v)`. The rationale behind the method signature is clarified in Theorem 3.1.

---

[1] This matches the output subtyping rule [$\leqslant_\ell$-Out] in Theorem 7.4.
[2] Such a requirement could be enforced by defining the field as `val`, instead of `def`; the drawback is that `val` does not allow type parameters, and this would result in an *in*variant `Out` with limited subtyping.
[3] This matches the input subtyping rule [$\leqslant_\ell$-In] in Theorem 7.4.

▶ **Example 3.1** (`!`, `?` and selection/branching)**.** Consider the following classes:

```
sealed abstract class AorB
case class A() extends AorB;   case class B() extends AorB
```

Let `c` be an instance of `Out[AorB]`. The `c.!` method can be used as follows:

```
c ! A()
```

or

```
c ! B()
```

Note that `!` resembles the output/selection operator seen in Fig. 4 (left). Moreover, the Scala compiler ensures that the argument of `!` belongs to a subtype of `AorB`, – e.g., `A` or `B`[4]: this corresponds, in session-π, to the type checking of an internal choice.

Let now `c` be an instance of `In[AorB]`. The `c.?` method can be used as shown below,

```
c ? { case A() => println("Got A")
      case B() => println("Got B") }
```

where the `{...}` block, as per usual Scala syntax, is a function from `AorB` to `Unit`. This reminds the branching operator seen in Fig. 4 (left). Moreover, since `AorB` is a `sealed abstract class`, the Scala compiler can check exhaustiveness, warning if the `case`s do not cover *both* `A` and `B`[5]: this corresponds, in session-π, to the type checking of an external choice.

Using `lchannels` endpoints: static vs. dynamic checks. As seen in Theorem 3.1, the Scala compiler can check that an instance of `lchannels` `Out` (resp. `In`) carrying a `sealed abstract class` is only used under the *safety* and *exhaustiveness* guarantees of a session-typed channel endpoint with a top-level $\oplus$ (resp. $\&$)[6], i.e., **S1** and **S2** in § 2.1. Also, an instance of e.g. `Unit` provides the guarantees of an **end**-typed channel endpoint: it cannot be used for I/O. Unfortunately, the Scala type checker cannot enforce *input/output linearity* (**S3** and **S4** in § 2.1); hence, `lchannels` implements the following *runtime linear usage rules*:

L1. each `Out` instance should be used to perform *exactly one* output. Any further output will generate a *runtime exception*, forbidding duplicated message transmissions;

L2. each `In` instance should be used *at least once*. Each use will *retrieve the same value*.

L1 and **L2** reflect the typical usage of Scala's `Promise`s and `Future`s. The lack of static linearity checks impacts deadlock-freedom guarantees: we will discuss this topic in § 6.1.3. Note that **L1** matches **S3**, while **L2** is more relaxed than **S4**. The latter is not a technical necessity, since `In` could be easily designed to raise an exception if used twice for input; we adhere to the familiar behaviour of `Future`s for simplicity of presentation, and to readily apply some common programming patterns, e.g. registering one or more input callbacks.

3.2 A local implementation

Fig. 6 (right) shows a simple *local* implementation of `In[A]`/`Out[A]`, as a thin layer over a `Promise[A]`/`Future[A]` pair (created in lines 12–14): a value written in the former becomes available on the latter. The `A`-cast in line 5 (due to the *in*variance of `Promise[A]`) is safe: the type bound on `B` ensures that `Promise[B]` can only be written with a subtype of `A`.

▶ **Example 3.2** (Spawning interacting threads)**.** Two threads that communicate through a local (linear) channel can be created with a method similar to the following:

[4] Due to Java legacy, in Scala also `Null` is a subtype of `AorB`. This will be explicit in § 7.3.
[5] By design, Scala does not enforce matching on `null` values, albeit they might be received (see note 4).
[6] This arises from the encoding of session types into linear I/O types with *variants* [8]: we render the latter in Scala as `sealed case class`es (as detailed in § 7.3).

```
1 case class Q(p: Boolean, cont: Out[R])
2 case class R(p: Int)
3
4 def f(c: In[Q]) = {
5   c ? { q => q.cont ! R(42) }
6 }
7
8 def g(c: Out[Q]) = {
9   val (ri,ro) = c.create[R]()
10  c ! Q(true, ro)
11  ri ? { r =>
12    println(f"Got ${r.p}")
13 } }
```

Figure 7 $S_{\mathtt{QR}}$ and $\overline{S_{\mathtt{QR}}}$ in Scala.

```
1 def parallel[A, B1, B2](p1: In[A] => B1, p2: Out[A] => B2): (Future[B1], Future[B2]) = {
2   val (in, out) = LocalChannel.factory[A]();  ( Future { p1(in) }, Future { p2(out) } )
3 }
```

Here, `p1` and `p2` are functions taking respectively an input and output channel endpoint carrying `A`, and returning resp. `B1` and `B2`. The `parallel` method creates a pair of `A`-carrying local channel endpoints (line 2), applies `p1` and `p2` on them by spawning separate threads, and returns a pair of `Future`s that will be completed with their return value (line 3).

Actually, `parallel` is a method of the `LocalChannel` object in Fig. 6. Most of the examples in the rest of the paper feature two endpoint functions with the signature of `p1` and `p2`, and they can be executed concurrently (and type-safely) via `LocalChannel.parallel`.

Our local implementation of `lchannels` is suitable for type-safe inter-thread communication, as suggested in Theorem 3.2. However, `Promise`/`Future` instances *cannot be serialised*, and thus cannot be sent/received over a network: this makes `LocalIn` and `LocalOut` unsuitable for *distributed* applications. We address this issue later on, in § 5.

4 Session programming with `lchannels` and CPS protocols

We now address Problem **P2** in § 2.3: given a session type S, how to instantiate the type parameters of `In[·]`/`Out[·]`, to represent the (possibly recursive) sequencing of internal/external choices of S. The answer lies in *representing the states of S as CPS protocol classes*, as outlined in § 2.4. We give an example-driven intuition of such a representation, and the resulting *session-based software development approach* (§ 4.2). The formalisation is in § 7.

4.1 Representing sequential inputs/outputs

Let us consider the session type $S_{\mathtt{QR}} = ?\mathtt{Q}(\mathsf{Bool}).!\mathtt{R}(\mathsf{Int})$, dictating that a channel endpoint must be used first to receive `Q`(Bool), and then to output `R`(Int). In Scala, we could define the two `case class`es on the right (where the field `p` stands for "payload"), and we can instantiate a linear input endpoint of type `In[Q]`, which allows to perform the first input of $S_{\mathtt{QR}}$; but, how do we require to send a value of type `R` along the same interaction?

```
1 case class Q(p: Boolean)
2 case class R(p: Int)
```

Inspired by the encoding of session types into linear types [8], we can *instead* define the case classes in Fig. 7 (lines 1–2), where `cont` stands for "continuation" (and recalls `replyTo` in § 1). Now, the value received from `In[Q]` also carries an `Out[R]` endpoint for continuing the interaction; the value received from `In[R]`, instead, does *not* have a `cont` field, since the

protocol ends there. In lines 4–6, `f` uses `c` to receive a `Q`-typed value `q` (line 5); then, uses `q.cont` to send a value of type `R`.

Now, consider the dual $\overline{S_{\texttt{QR}}}$ = `!Q(Bool).?R(Int)`: we can represent it in Scala simply by *reusing* `Q` *and* `R` in Fig. 7, and instantiating a linear *output* endpoint `Out[Q]`. Its usage is shown in lines 8–13. To produce a value of type `Q`, `g` must also produce a channel endpoint `Out[R]`: for this reason, the two continuation endpoints `ri`,`ro` are created (line 9), respectively with types `In[R]`,`Out[R]`. On line 10, `c` is used to send a `Q`-typed value, carrying `ro`: the recipient is expected to use it for continuing the interaction; on line 11, `ri` is used to receive the value `r` (of type `R`) sent on `ro`.

4.2 A development approach for session-based applications

In our last example, `Q` and `R` are the *CPSP classes* of both $S_{\texttt{QR}}$ and $\overline{S_{\texttt{QR}}}$, `In[Q]` is the Scala representation of $S_{\texttt{QR}}$, while `Out[Q]` is the representation of $\overline{S_{\texttt{QR}}}$. We can outline a *development approach for session-based applications*. For each communication channel:

D1. formalise the two endpoint session types S and $\overline{S}$ (assuming they are not trivially **end**);
D2. extract the CPSP classes of S (or, equivalently, of $\overline{S}$). Roughly, it means:

- **a.** convert each internal/external choice into a set of `case class`es (one per label);
- **b.** when a choice has multiple labels, let each `case class` above extend a common `sealed abstract class`, representing the multiple choice itself;
- **c.** recover the sequencing in S (and $\overline{S}$) by "connecting" each `case class` to its "successor" (if any), through the `cont` field;

D3. let `C` be the class representing the outermost internal/external choice of S:

- if S starts with an internal choice, its Scala endpoint type is `Out[C]`. Dually, since $\overline{S}$ starts with an external choice, the Scala type at the other endpoint is `In[C]`;
- otherwise, if S starts with an external choice, its Scala endpoint type is `In[C]`. Dually, since $\overline{S}$ starts with an internal choice, the Scala type at the other endpoint is `Out[C]`.

The extraction of protocol classes must deal with some subtleties, in particular for determining whether `cont` should be an `In[·]` or `Out[·]` endpoint, and for representing recursion. We will formally address these issues in § 7.3; now, we proceed with more examples.

4.3 Interlude: automating channel creation

```
1  abstract class Out[-A] { ...
2    def !![B](h: Out[B] => A): In[B] = {
3      val (cin, cout) = this.create[A]()
4      this ! h(cout)
5      cin
6    }
7    def !![B](h: In[A] => B): Out[B] = {
8      val (cin, cout) = this.create[A]()
9      this ! h(cin)
10     cout
11 } }
```

Before proceeding, we take a quick detour to address Problem **P3** of § 2.3. In Fig. 7 (line 9), we can notice a case of *manual creation of channel endpoints*, as in Fig. 4 (right, line 3). This is a key pattern for "CPS interactions": when sending a message that does *not* conclude a session, it is necessary to *create* a pair of channels, *send* one of them, and use the other to *continue* interacting[7]. This "create-send-continue" pattern ensures session progress, but is an error-prone burden for the programmer; so, we automate it by extending `Out` (Fig. 6, left) with the method `!!` above.

Take `c` of type `Out[Q]` from Fig. 7 (lines 8–13), and let `h` be a function from `Out[R]` to `Q`: `c !! h` *creates* a pair of channel endpoints `(cin,cout)` of type `In[R]`,`Out[R]` (line 3 above),

[7] The pattern actually reflects how session-π processes are encoded in standard π-calculus (§ 2.3).

applies `h` to `cout`, *sends* the result via `c` (line 4), and returns `cin` for *continuing* the session (the other case of `!!` is "dual", when `h`'s domain is `In[R]`). By letting `h` be an instance of `Q` with a hole in place of `cont`, we can remove line 9 of Fig. 7, and rewrite line 10 as:

```
1 val ri = c !! Q(true, _:Out[R])
```

, where the type annotation is necessary due to the limited type inference capabilities of Scala[8].

We can address this last inconvenience by defining `Q` as a *curried* `case class`, and placing the hole in the curried `cont` field: the Scala compiler can now infer its type. The resulting code is shown on the right (with `f` unchanged w.r.t. Fig. 7). We will adopt this style for the rest of the paper.

```
1 case class Q(p: Boolean)
2             (val cont: Out[R]) // Curried
3 case class R(p: Int)
4
5 def g(c: Out[Q]) = {
6   val ri = c !! Q(true)_ // No type annot.
7   ri ? { r => println(f"Got ${r.p}") }
8 }
```

4.4 Examples

We now discuss some examples of the session-based approach outlined in § 4.2. We proceed by increasing complexity, showing how to instantiate CPSP classes to represent recursion (Theorem 4.1), non-singleton external/internal choices (Theorem 4.2), and multiple channels with *higher-order* types for *session delegation* (Theorem 4.3).

▶ **Example 4.1** (FIFO). An unidirectional FIFO channel, with endpoints for sending/receiving values of type T, can be represented with the following recursive session types:

$$S_{\text{fifo}} = \mu X.!\texttt{Datum}(T).X \text{ (sending endpoint)} \qquad \overline{S_{\text{fifo}}} = \mu X.?\texttt{Datum}(T).X \text{ (receiving endpoint)}$$

The corresponding CPSP classes consist in just one (parametric) declaration:

```
1 case class Datum[T](p: T)(val cont: In[Datum[T]])
```

i.e., we represent the recursion on X by *(i)* taking the name of the `class` corresponding to the outermost internal/external choice under $\mu X.\ldots$ (i.e., `Datum`), and *(ii)* `cont`inuing with such a name when X occurs (for another case of recursion, see Theorem 4.2). Note that `cont` is an *input* endpoint, used by the *recipient* to receive a further value, while the *sender* keeps the output endpoint to produce a value. The endpoint processes can be written as:

```
1 def sender(fifo: Out[Datum[Int]]): Unit = {
2   val cont = fifo !! Datum(1)_ !! Datum(2)_
3   sender(cont)
4 }
```

```
1 def receiver(fifo: In[Datum[Int]]): Unit = {
2   val v = fifo.receive
3   println(f"Got ${v.p}");  receiver(v.cont)
4 }
```

Here, `sender` performs two outputs in a row (line 2): this is allowed since each application of `!!` returns a channel of type `Out[Datum[T]]` (cf. declaration of `Datum[T]` above).

▶ **Example 4.2** (Greeting protocol). Consider the "greeting" types S_{h} and $\overline{S_{\text{h}}}$ from § 2. Unlike Theorem 4.1, we now have *non-singleton* internal/external choices. To extract their CPSP classes, we apply item **D2**b of § 4.2: *add a* `sealed abstract class` *for each internal/external choice*, extending it with one `case class` per label. In this case, we add:

- `Start` for the internal choice of S_{h} (i.e., the external choice of $\overline{S_{\text{h}}}$) between `Greet`,`Quit`;
- `Greeting` for the external choice of S_{h} (i.e., the internal choice of $\overline{S_{\text{h}}}$) between `Hello`,`Bye`.

[8] This limitation is present in Scala 2.11.8, but might be overcome in future versions.

We obtain the CPSP classes on the right, with `Out[Start]`/`In[Start]` representing $S_h/\overline{S_h}$ (by **D3**). We can write two endpoint processes as:

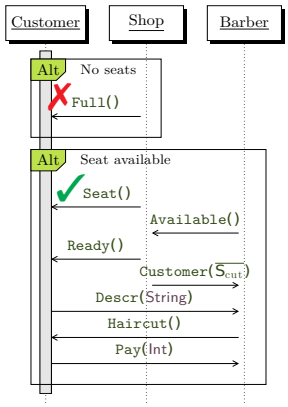
```
sealed abstract class Start
case class Greet(p: String)(cont: Out[Greeting]) extends Start
case class Quit(p: Unit)                         extends Start

sealed abstract class Greeting
case class Hello(p: String)(cont: Out[Start]) extends Greeting
case class Bye(p: String)                     extends Greeting
```

```
def client(c: Out[Start]): Unit = {
  if (Random.nextBoolean()) {
    val c2 = c !! Greet("Alice")
    c2 ? {
      case m @ Hello(name) => client(m.cont)
      case Bye(name) => ()
    }
  } else { c ! Quit() } }
```

```
def server(c: In[Start]): Unit = {
  c ? {
    case m @ Greet(whom) => {
      val c2in = m.cont !! Hello(whom)
      server(c2in)
    }
    case Quit() => ()
} }
```

Note that `client` is similar to the pseudo code of `hello` in Fig. 4 (left).

▶ **Example 4.3** (Sleeping barber with session delegation)**.** We address a classical problem in concurrency theory [10]: a barber waits for customers in his shop, sleeping when there is nobody to serve. When a customer enters in the shop, he goes through a waiting room with n chairs: if all chairs are taken, he leaves; otherwise, he sits. If the barber is sleeping, he wakes up, serves all sitting customers (one a time), and sleeps again when nobody is waiting. We model this scenario with three components: the customer, the shop and the barber, using session types to formalise their expected interactions, schematised below.



In this example, we show how multiple concurrent sessions (one per customer) can be handled by single-threaded programs (shop and barber). We also show how to exploit *session delegation* by leveraging *higher-order* session types (i.e., channel endpoints that send/receive other channel endpoints). When a customer enters in the shop, he gets a S_{cstm}-typed channel endpoint:

$$S_{\text{cstm}} = ?\texttt{Full} \,\&\, ?\texttt{Seat}.?\texttt{Ready}.S_{\text{cut}} \qquad S_{\text{cut}} = !\texttt{Descr}(\texttt{String}).?\texttt{Haircut}.!\texttt{Pay}(\texttt{Int})$$

He might receive either a `Full` message (when no seats are available), or a `Seat`: in the first case, the session ends; in the second case, he waits for the barber to be `Ready`. Then, he continues with S_{cut}: `Descr`ibes the new hairdo, waits for the `Haircut`, `Pay`s and leaves. The shop uses the other, dually-typed channel endpoint:

$$\overline{S_{\text{cstm}}} = !\texttt{Full} \oplus !\texttt{Seat}.!\texttt{Ready}.\overline{S_{\text{cut}}} \qquad \overline{S_{\text{cut}}} = ?\texttt{Descr}(\texttt{String}).!\texttt{Haircut}.?\texttt{Pay}(\texttt{Int})$$

and keeps track of the n seats to choose whether to send `Full` or `Seat`. When the customer gets a `Seat`, the shop interacts with the barber, through a channel with endpoint types:

$$S_{\text{barber}} = \mu X.!\texttt{Available}.?\texttt{Serve}(\overline{S_{\text{cut}}}).X \quad \text{(barber endp.)} \qquad \overline{S_{\text{barber}}} = \mu X.?\texttt{Available}.!\texttt{Serve}(\overline{S_{\text{cut}}}).X \quad \text{(shop endp.)}$$

i.e., the shop recursively waits for the barber to be `Available`; when it happens, it picks a sitting customer (i.e., one that has received a `Seat`), sends a `Ready` message to him, and forwards the channel endpoint (now $\overline{S_{\text{cut}}}$-typed) to the barber, as the payload of `Serve`.

Meanwhile, the barber uses its S_{barber}-typed channel endpoint to notify that he is `Available`, and wait for a `Serve` message – sleeping until he gets one; when it happens, the barber gets a $\overline{S_{\text{cut}}}$-typed channel endpoint in the message payload: he is expected to use it for interacting with the customer, i.e., listen for the hairdo `Descr`iption, perform the `Haircut`, and take the `Pay`ment. When the customer session terminates, the barber must resume his recursive session with the shop: he notifies that he is `Available` again, *etc.*

The CPSP classes extracted from the session types above are shown on the right. As per item **D2**b of § 4.2, we introduce `WaitingRoom` as the `sealed abstract class` corresponding to the external (resp. internal) choice between `Full` and `Seat` in S_{cstm} (resp. $\overline{S_{\text{cstm}}}$).

```
1  // Customer <--> shop protocol
2  sealed abstract class WaitingRoom
3  case class Full()                          extends WaitingRoom
4  case class Seat()(val cont: In[Ready])     extends WaitingRoom
5
6  case class Ready()(val cont: Out[Description])
7  case class Description(p: String)(val cont: Out[Cut])
8  case class Cut()(val cont: Out[Pay])
9  case class Pay(p: Int)
10
11 // Barber <--> shop protocol
12 case class Available()(val cont: Out[Serve])
13 case class Serve(p: In[Description])(val cont: Out[Available])
```

Implementation. The code of the shop, barber and customer is shown in Fig. 8. They are supposed to run as concurrent threads, and thus implement the `Runnable` interface.

`Shop` is parametric in the number of seats. It collects the channel endpoints of the waiting customers in its private `seats` field, which may be any FIFO-like container with a *blocking* `read` method: we could use e.g. `scala.concurrent.Channel[Out[Ready]]`, or a FIFO based on Theorem 4.1. Once started, `Shop` creates a S_{barber}-typed channel (line 19) and gives the output endpoint to a new `Barber` (line 20). The `enter` method returns an input endpoint for interacting according to S_{cstm}: after creating two channel endpoints of the suitable type (line 6), `enter` checks how many people are trying to get a seat, and outputs `Full` (line 10) or `Seat` (line 12) *before* returning the input endpoint (line 15). In the main `loop` (lines 24–33), the shop waits for an `Available` message from the barber (line 25), sleeps while retrieving a customer channel from `seats` (line 26), notifies the customer that the barber is `Ready`, forwards the channel to the barber, and continues its `loop`.

`Barber`, in line 7, notifies the shop that he is `Available`, and uses the channel endpoint returned by `!!` (whose type is `In[Serve]`) to wait for a `Serve` message. Then, he interacts with the customer using the `In[Descr]`-typed endpoint received as payload (lines 8–11); after being paid, he continues the session with the shop (line 11).

The code for `Customer` is simple: he invokes the `enter` method of the `Shop` given as parameter (line 3), and uses the returned channel to interact according to S_{cstm}. If the waiting room is `Full`, he retries later (lines 5–7). To model multiple customers competing for the seats, it is sufficient to start multiple `Customer`s referring to the same `Shop`.

As anticipated, our solution for the sleeping barber problem exploits *session delegation*: the customer starts interacting with the shop, but his session is eventually forwarded to the barber, with a higher-order `Serve`($\overline{S_{\text{cut}}}$) message. Delegation is *transparent*: no dedicated code is required in `Customer`'s implementation. Moreover, delegation is safe: e.g., the Scala type checker ensures that only `Out[Ready]`-typed channel endpoints are stored in `Shop.seats`, and that the barber picks up the session only after the shops sends `Ready`.

5 Optimisations, transport abstraction and error handling

In this section, we demonstrate how `lchannels` allows to abstract from the underlying message transport medium, and to handle communication errors. In § 3, we introduced the abstract classes `In`/`Out`, and `LocalIn`/`LocalOut` as simple *local* implementations for inter-thread communication. The `In[·]`/`Out[·]` interface can abstract other message transports, allowing `lchannels`-based programs to achieve faster message delivery, or transparently interact across a network. We discuss 3 examples: queue-, actor- and stream-based channels.

Optimised queue-based channels. The simple `LocalIn`/`LocalOut` classes in Fig. 6 (right) perform all communications through the underlying `Future`/`Promise`. However, many

```
class Shop(nSeats: Int) extends Runnable {
  private val seats: Fifo[Out[Ready]] = Fifo() // Customers queue
  private val waiting = new AtomicInteger(0) // Customers in shop

  def enter(): In[WaitingRoom] = {
    val (in, out) = LocalChannel.factory[WaitingRoom]()
    val nPeople = waiting.getAndIncrement() // New person in shop
    if (nPeople >= nSeats) { // More people than seats
      waiting.getAndDecrement() // Customer must leave
      out ! Full() // Tell customer that the witing room is full
    } else {
      val r = out !! Seat()_ // Tell customer that he got a seat
      seats.write(r) // Add customer to waiting queue
    }
    in // Return input endpoint of customer channel
  }

  override def run(): Unit = {
    val (bIn, bOut) = LocalChannel.factory[Available]()
    new Barber(bOut).start() // Spawn barber with output endpoint
    loop(bIn) // Loop on the input endpoint
  }

  private def loop(bIn: In[Available]): Unit = {
    bIn ? { avl => // The barber is available
      val cust = seats.read // Take 1st customer, sleep if none
      waiting.getAndDecrement() // Customer is leaving the seat
      val cust2 = cust !! Ready()_ // Notify that barber is ready
      // Forward the customer to the barber
      val bIn2 = avl.cont !! Serve(cust2)_ // bIn2: In[Available]
      loop(bIn2) // Keep interacting with the barber
    }
  }
}
```

```
class Barber(c: Out[Available]) extends Runnable {
  override def run(): Unit = {
    loop(c)
  }

  private def loop(c: Out[Available]): Unit = {
    (c !! Available()_) ? { srv => // Got customer
      val d = srv.p.receive // Got haircut descr
      val payC = d.cont !! Haircut()_ // Cut hair
      val pay = payC.receive // Wait payment
      // Got pay, no continuation: customer done
      loop(srv.cont) // Continue shop interaction
    }
  }
}
```

```
class Customer(shop: Shop) extends Runnable {
  override def run(): Unit = {
    val s = shop.enter() // Type: In[WaitingRoom]
    s ? {
      case Full() => { // No seats
        Thread.sleep(...) // Random wait
        run() // Try taking a seat again
      }
      case m @ Seat() => { // Got a seat
        val r = m.cont.receive // Barber ready
        val cutC = r.cont !! Descr("Fancy cut")_
        val cut = cutC.receive // Wait for cut
        cut.cont ! Pay(42) // Cut done, paying
      }
    }
  }
}
```

Figure 8 Sleeping barber (Theorem 4.3): shop, barber and customer implementations.

applications could mostly use the `In.receive`/`Out.send` methods, and could benefit from an optimised implementation of `In`/`Out` that (when possible) bypasses `In.future`/`Out.promise`. We developed this idea with the `QueueIn`/`QueueOut` classes: internally, they deliver messages through Java `LinkedTransferQueue`s (under the runtime linearity constraints **L1**/**L2** of §3.1) – and only allocate and use a `Future`/`Promise` when the `.future`/`.promise` methods are *explicitly* invoked. Moreover, we optimised the `QueueOut.!!` method to reuse queues when continuing a session. The resulting performance improvements are shown in § 6.2.

Network-transparent actor-based channels. We implemented proof-of-concept *network-transparent* subclasses of `In`/`Out`, called `ActorIn`/`ActorOut`: they deliver messages by automatically spawning *Akka Typed* actors [29], which in turn can communicate over a network.

Using such actor-based channels, a local process can interact with a remote one through a *local* actor-based endpoint that proxies a *remote* endpoint. E.g., to obtain a remote interaction between greeting `server` and `client` (Theorem 4.2) we can run the former as:

```
val (in, out) = ActorChannel.factory[Start]("start");   server(in)
```

Now, `out.path` contains the Akka Actor Path [27] of an automatically-generated actor. Such a path can be used, *even on a different JVM*, to instantiate a proxy for `out`, as follows:

```
val c = ActorOut[Start]("akka.tcp://sys@host.com:5678/user/start");   client(c)
```

where `ActorOut`'s argument matches `out.path` above. Then, the `client` and `server` will interact over a network, without changing their code.

All the examples in this paper can also run on `ActorChannel`s, simply by replacing the calls to `LocalChannel.factory[A]()` with `ActorChannel.factory[A]()` (e.g. in Fig. 8, `Shop`, line 6). To achieve complete transport-independence, `factory` can be parameterised.

We choose Akka as a message transport medium due to its widespread availability, using Akka Typed to obtain stronger static typing guarantees throughout the implementation. The main challenges were related to making `ActorIn`/`ActorOut` instances *serializable*: this

is a crucial requirement, as channel endpoints might appear (as payloads or continuations) in messages sent/received over a network. In particular, sending an `ActorOut[A]` roughly corresponds to sending an `ActorRef[A]` instance (which is serializable out-of-the-box) – but sending an `ActorIn[A]` has no Akka equivalent, and requires some internal machinery.

Network-transparent stream-based channels. Often, programs interacting over a network are implemented with different languages, and use bare TCP/IP sockets without a common higher-level networking framework. Still, such programs might need to observe complicated protocols (e.g. RFC-based ones like POP3, SMTP, *etc.*) that can be abstractly represented as session types [12, 21]. To address this scenario, we extended `lchannels` with channel endpoints that send/receive messages through Java `InputStream`/`OutputStream`s, obtained e.g. from a network socket. The main classes are `StreamIn`/`StreamOut` (extending resp. `In`/`Out`), and can only be instantiated by providing a protocol-specific `StreamManager` which can serialize/deserialize messages to/from a stream of bytes (tracking the session status if needed).

Message	Text format
`Greet("Alice")`	`GREET Alice`
`Hello("Alice")`	`HELLO Alice`
`Bye("Alice")`	`BYE Alice`
`Quit()`	`QUIT`

For example, suppose that the "greeting protocol" from Theorem 4.2 abstracts a textual protocol as shown on the left, and we want our `client` to interact with a third-party server using that textual format over TCP/IP sockets. We first need to derive the `StreamManager` class, implementing a `HelloStreamManager` that suitably serializes/deserializes the textual messages. Then, we can let our `client` talk with a remote server, via TCP/IP, using the textual format:

```
val conn = new Socket("host.com", 1337) // Hostname and port where greeting server runs
val strm = new HelloStreamManager(conn.getInputStream, conn.getOutputStream)
val c = StreamOut[Start](strm) // Output channel endpoint, towards host.com:1337
client(c)
```

Note that we did not change the code of `client` seen in Theorem 4.2: we leverage `lchannels` and protocol classes to represent and type-check the high-level protocol structure (sequencing, choices, recursion), while separating the low-level details from the logic of the program.

Error handling. The methods of `In[A]` seen in Fig. 6 do not handle errors; e.g., `receive` throws an exception if no message arrives within the (implicit) `Duration d`. However, input errors are quite common in real-world applications: e.g., the process at the other endpoint might not timely send a message, or may send a wrong message that a `StreamManager` cannot deserialize, or a network problem may occur. As typical for Scala APIs, we extended `In[A]` to capture failures as `Try[A]` values, via 2 additional methods: `tryReceive` and `??`.

```
c ?? { case Success(m) => m match {
         case A() => println("Got A")
         case B() => println("Got B") }
       case Failure(e) => /* Inspect e */ }
```

E.g., the branching on `AorB` in Theorem 3.1 can be made error-resilient by using `c.??`, as shown on the left: the top-level matching is now on `Try[AorB]`.

6 Evaluation

We now assess the practicality of the approach in § 4.2 with a case study based the "client with frontend" in Fig. 1 (§ 6.1), and a performance evaluation of `lchannels` (§ 6.2).

6.1 A case study: application server with frontend

This section shows how our approach can address the "server with frontend" scenario in § 1. We consider an application server that is a *chat server* allowing users to join/leave chat rooms, and send/receive messages to/from them. We formalise the protocols of the

application (§ 6.1.1), and illustrate some characteristics of the implementation (§ 6.1.2), and discuss how development was aided by CPS protocols and `lchannels` (§ 6.1.3).

6.1.1 The protocols

We formalise the protocols in Fig. 1 as session types, dividing them in two groups: *public* (used by clients), and *internal* (used for frontend/auth/chat server interaction).

Public protocols. The session type S_{front} formalises the usage of the channel endpoint that the frontend handles while interacting with a client. It is defined as follows:

$$S_{\text{front}} = ?\texttt{GetSession}(\mathsf{Id}).\big(!\texttt{New}(S_{\text{auth}}) \oplus !\texttt{Active}(S_{\text{act}})\big) \quad S_{\text{auth}} = !\texttt{Authenticate}(\mathsf{Cred}).\big(?\texttt{Success}(S_{\text{act}}) \,\&\, ?\texttt{Failure}\big)$$
$$S_{\text{act}} = \mu X.\Big(!\texttt{Quit} \oplus !\texttt{GetId}.?\texttt{Id}(\mathsf{Id}).X \oplus !\texttt{Ping}.?\texttt{Pong}.X \oplus !\texttt{Join}(\mathsf{String}).?\texttt{ChatRoom}((S_{\text{r}}, S_{\text{rctl}})).X\Big)$$

The service implementing S_{front} waits for a `GetSession(Id)` request from a client; then, with an internal choice $\oplus$, it might answer by sending either `New`(S_{auth}) or `Active`(S_{act}):

- `New` carries a S_{auth}-typed channel endpoint, talking with the auth server: it allows the client to send an `Authenticate(Cred)` message (with `Cred` being the credentials), and wait for either `Success`(S_{act}) or `Failure` (the S_{act}-typed channel is explained below);
- `Active` carries an S_{act}-typed channel endpoint representing the active "session loop" (Fig. 1). When the client receives it, S_{act} (which is recursive) allows to choose among:
  - `Quit`. In this case, the chat session ends;
  - `GetId`. Then, the client receives an `Id(Id)` answer whose payload is the current session identifier, and continues the session recursively;
  - `Ping(String)`. Then, the client receives a `Pong(String)`, and continues recursively;
  - `Join(String)`, with the payload being a chat room name. Then, the client joins a chat room, gets a `ChatRoom`$((S_{\text{r}}, S_{\text{rctl}}))$ answer, and the session continues recursively. The two channels endpoints in the payload allow to interact with the chat room:
    - $S_{\text{r}} = \mu Y.?\texttt{NewMessage}((\mathsf{String}, \mathsf{String})).Y \,\&\, ?\texttt{Quit}$. This recursive endpoint allows the client to receive either a `NewMessage` from the chat room (with the payload being the sender username and the message text), or `Quit` (ending the interaction);
    - $S_{\text{rctl}} = \mu Z.!\texttt{SendMessage}(\mathsf{String}).Z \oplus !\texttt{Quit}$. This endpoint allows the client to send either a message on the chat room (with the payload being the text), or `Quit`.

The CPS protocol classes of the session types above are extracted as in the examples of § 4.4, and are almost identical to Fig. 2. In particular, we use `Command` as the `sealed abstract class` for the top-level choice in S_{act} (this detail will be mentioned again in § 6.1.2).

Internal protocols. Fig. 1 also outlines the *internal* communications among the frontend, authentication and chat server: they can be formalised as session types, too – as for barber-shop interaction in Theorem 4.3. Here, we only detail the frontend-server interaction type:

$$S_{\text{FS}} = \mu X.!\texttt{GetSession}(\mathsf{Id}).\big(?\texttt{Success}(S_{\text{act}}).X \,\&\, ?\texttt{Failure}.X\big)$$

The frontend recursively queries for active sessions (passing the Identifier received from a client), getting either `Success` or `Failure`. In the first case, the message payload is a S_{act}-typed channel endpoint, that will be forwarded to the client with an `Active` message.

6.1.2 The implementation

This case study uses *higher-order session types* to naturally model the "handles" mentioned in § 1. A difference w.r.t. Theorem 4.3 is that the delegation appears *explicitly* in client's session types, e.g. in `Active` messages with a channel as payload. In CPS protocols, this difference is almost negligible: the `Active` message classhas no `cont`inuation, but the client should keep interacting via the `Out` endpoint in the payload – as per rule **L1** in § 3.1.

The server-side implementation reuses several solutions from Theorem 4.3 – e.g., internal FIFOs for storing and later processing requests: this happens e.g. when the single-threaded chat server manages multiple client sessions. The main difference w.r.t. Theorem 4.3 is that requests are queued *asynchronously* (via `In.future`) and enriched with internal data.

```
1  class ChatServer(...) extends Runnable {
2    ...
3    private def createSession(username: String): Out[Command]) = {
4      val id = allocUniqueSessionId()
5      val (in, out) = LocalChannel.factory[Command]()
6      in.future.onComplete { // Using scala.util.{Success, Failure}
7        case Success(cmd) => queueRequest(Success((id, cmd)))
8        case Failure(e) => queueRequest(Failure(e))
9      }
10     // Add the new session to the list of known sessions
11     sessions(id) = ... /* session info, including username */
12     out
13 } }
```

E.g., the chat server calls the method on the left when the auth server asks to create a new session for `username`: it reserves a session `id` (line 4), creates the channel endpoints `in`,`out` carrying a `Command` (line 5), keeps `in`, and returns `out` (line 12), that will be the payload of a `NewSession` message. The client `Command` is received *asynchronously* via `in.future`: in lines 6–9, `cmd` is paired with the session `id`, and queued (line 7). When the pair is later dequeued and processed, `id` tells on which session `cmd` is acting. A similar queuing is performed as the session progresses; e.g., when a `cmd` of type `Ping` is dequeued, the server runs:

```
1  val in2 = cmd.cont !! Pong(cmd.msg)_ // cmd's type: Ping; in2's type: In[Command]
```

and `in2.future` is used for queuing the next client command, like `in.future` in lines 6–9.

6.1.3 Lessons learned

As expected, CPS protocols and `lchannels` allow the Scala type checker to detect protocol errors that usually arise on untyped channels, e.g., trying to send the wrong type of message, or forgetting to consider some cases when branching with `In.?`. This greatly simplified the present case study, where multiple channels with various protocols are handled concurrently. Since we leverage the *existing* Scala type system, modern Scala IDEs (such as [30]) provide channel usage errors and hints, e.g. via typing information and auto-completion suggestions.

However, as seen in § 3.1, Scala and `lchannels` cannot perform *static* linearity checks: hence, they cannot spot two kinds of errors, illustrated below, that impact session progress.

Double usages of output endpoints. They occur when an `Out[A]` instance is used twice to send `A`-typed values: then, by **L1** in § 3.1, an exception is thrown, and the extra message is *not* sent. This kind of error never occurred in our experience: the CPS interaction guided by `lchannels` seems to naturally shape programs where output endpoints are discarded after used. Moreover, as for Scala `Promise`s, double outputs causes an *immediate* runtime error, that (we believe) should usually arise in proximity of the code requiring a fix.

Unused channel endpoints. Not performing an output can leave a process at the other endpoint stuck, waiting for input – and this could escalate to other processes waiting on other channels; this problem can also arise if a program does not *input* a message whose continuation/payload is an *output* channel. Spotting this kind of errors can be tricky, especially if channels are dynamically generated, sent, received, stored in collections (as in our case study). `lchannels` mitigates this issue via timeouts on the receiving side

(§ 5): they allow to see which channel is stuck in which state – and thus, which process is not producing an output. In our case study, a few issues of this kind were easily fixed.

6.2 Benchmarks

We implemented several micro-benchmarks to evaluate how `lchannels` impacts communication speed w.r.t. other inter-thread communication methods: Fig. 9 shows the results. The benchmarks are mainly inspired by [24]; *"Streaming"* is a parallel blend of *"Ring"*+*"Counting actor"*: 16 threads are connected in a ring and a sequence ("stream") of messages is sent *at once*, measuring the time required for *all* to complete one loop.

We wrote an implementation of each benchmark using `Out.send`/`In.receive` for inter-thread communication, instantiating them with `LocalChannel`s, `QueueChannel`s and `ActorChannel`s (columns 1, 5, 7). As a comparison, we adapted such implementations to interact via `Promise`s/`Future`s (column 2), and also to interact "non-CPS" via `scala.concurrent.Channel`s, and Java `ArrayBlockingQueue`s / `LinkedTransferQueue`s (columns 3, 4, 6).

The overhead of `lchannels` w.r.t. "non-CPS" queue-based interaction has two origins:

1. *runtime linearity checks*, i.e. inspecting/setting a flag when a channel endpoint is used;
2. *repeated creation of* `In`/`Out` *continuation pairs* (§ 4.3): in comparison, our "non-CPS" benchmarks create Scala channels / Java queues just *once* at the beginning of each session.

Hardware/JVM settings highly influence the measurements: queues or `Promise`s/`Future`s can become relatively faster/slower, or show more/less variance, depending on the benchmark. Still, the results tend to be consistent with Fig. 9. It can be seen that `LocalChannel`s add a small slowdown to the underlying `Promise`s/`Future`s. `QueueChannel`s are considerably faster, *except* when many short-lived sessions are rapidly created (this scenario is stressed by "Chameneos", against the optimisations seen in § 5); still, `QueueChannel`s add a perceivable overhead on the underlying `LinkedTransferQueue`s. `ActorChannel`s are slower, especially with many threads and low parallelism (as in "Ring"): it is due to the (currently unoptimised) internal machinery that makes `ActorChannel`s *network-transparent*, and more suitable for *distributed* settings where network latency can make the slowdown less relevant.

Notably, the usual "non-CPS" communication we implemented (and measured) over Scala channels / Java queues requires connecting pairs of threads P_1,P_2 with pairs of queues (one carrying messages from P_1 to P_2, the other from P_2 to P_1). Such queues have type `Queue[A]`, where `A` must cover *all* the message types that could be sent/received: for protocols with sequencing and branching, this leads to loose static type checks, that combined with the lack of runtime monitoring, increase the risk of protocol violations errors.

7 A formal foundation

We now explain the formal foundations of our approach (as outlined in § 4.2), by detailing how to extract CPSP classes from session types, and studying how Scala's type system handles session subtyping/duality. We summarise *session subtyping* (§ 7.1), and we introduce our encoding from session to linear types (§ 7.2), and then into Scala types (§ 7.3).

7.1 Session types and subtyping

We defined session types and duality in § 2; to ease the treatment, we adopt 2 restrictions.

▶ Remark 7.1 (Syntactic restrictions). For all S, *(i)* each label is unique, and also a valid Scala class name, and *(ii)* each μ binds a *distinct* variable that actually occurs in its scope.

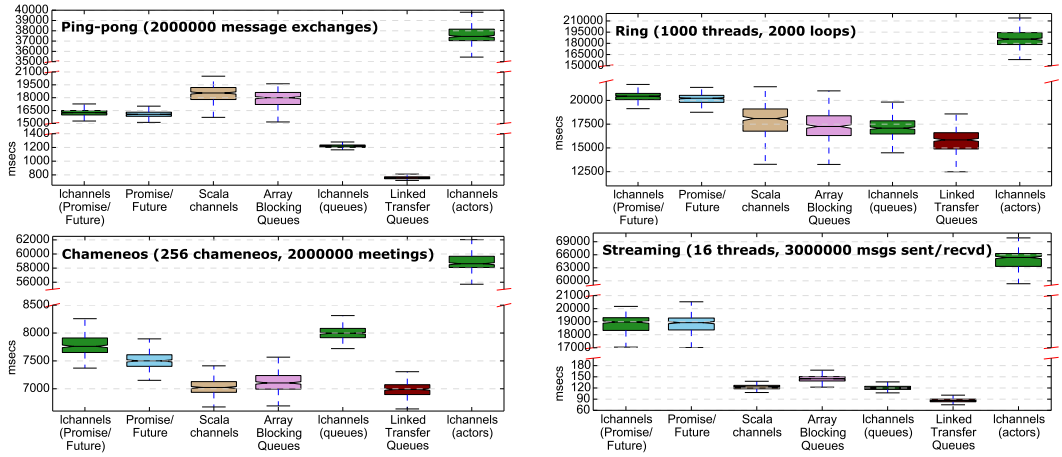


Figure 9 Benchmark results (box&whisker plot): 30 runs × 10 JVM invocations, Intel Core i7-4790 (4 cores, 3.6 GHz), 16 GB RAM, Ubuntu 14.04, Oracle JDK 64-bit 8u72, Scala 2.11.7, Akka 2.4.2.

Restriction *(i)* allows to directly generate a Scala `case class` from each internal/external choice label. Restriction *(ii)* is a form of Ottmann/Barendregt's variable convention [4].

The *session subtyping relation* $\leqslant$ allows to safely replace a S'-typed channel endpoint with a S-typed one, provided that $S \leqslant S'$ holds. The relation is defined as follows.

▶ **Definition 7.2** (Session subtyping [13])**.** The *subtyping relation between session types* is coinductively defined by the following rules (where $\leqslant_{\mathbb{B}}$ is a subtyping between basic types):

$$\frac{\forall i \in I\colon\quad T_i \leqslant T_i' \quad S_i \leqslant S_i'}{\&_{i\in I}?\mathtt{l}_i(T_i).S_i \;\leqslant\; \&_{i\in I\cup J}?\mathtt{l}_i(T_i').S_i'}\ [\leqslant\text{-Ext}] \qquad \frac{\forall i \in I\colon\quad T_i' \leqslant T_i \quad S_i \leqslant S_i'}{\oplus_{i\in I\cup J}!\mathtt{l}_i(T_i).S_i \;\leqslant\; \oplus_{i\in I}!\mathtt{l}_i(T_i').S_i'}\ [\leqslant\text{-Int}]$$

$$\mathbf{end} \leqslant \mathbf{end}\ [\leqslant\text{-End}] \qquad \frac{S\{\mu_X.S/X\} \;\leqslant\; S'}{\mu_X.S \;\leqslant\; S'}\ [\leqslant\text{-}\mu\mathrm{L}] \qquad \frac{S \;\leqslant\; S'\{\mu_X.S'/X\}}{S \;\leqslant\; \mu_X.S'}\ [\leqslant\text{-}\mu\mathrm{R}] \qquad \frac{T \leqslant_{\mathbb{B}} T'}{T \leqslant T'}\ [\leqslant\text{-}\mathbb{B}]$$

Rule [⩽-Ext] says that an external choice S is smaller than another external choice S' iff S offers a *subset* of the labels, and for all common labels, the payload and continuation types are in the relation. The rationale is that a program which correctly uses an S'-typed channel endpoint supports all its inputs – hence, the program also supports the more restricted inputs of an S-typed endpoint. Dually, [⩽-Int] says that an internal choice S is smaller than another internal choice S' iff S offers a *superset* of the labels, and for all common labels, the payload and continuation types are in the relation. The rationale is that a program which correctly uses an S'-typed channel endpoint might only perform one of the allowed outputs, that is also allowed by the more liberal S-typed endpoint. [⩽-End] says that a terminated session has no subtypes. [⩽-μL] and [⩽-μR] are standard. [⩽-$\mathbb{B}$] extends $\leqslant$ to basic types.

7.2 Linear I/O types (with records and variants)

In order to encode session types into Scala types, we exploit an intermediate encoding into *linear types for input and output [36]*. We focus on a subset of such types, defined below.

▶ **Definition 7.3.** Let $\mathbb{B}$ be a set of *basic types* (§ 2). A *linear type* L is defined as:

$$L ::= ?(U) \mid !(U) \mid \bullet \qquad U ::= [\mathtt{l}_i\_\{\mathtt{p} : V_i, \mathtt{c} : L_i\}]_{i\in I} \mid \mu_X.U \mid X \qquad V ::= \mathbb{B} \mid L \text{ (closed)}$$

where *(i)* recursion is guarded, and *(ii)* all $\mathtt{l}_i$ range over pairwise distinct labels. We also define the *carried type of* L as $\mathrm{carr}(?(U)) = \mathrm{carr}(!(U)) = U$.

$?(U)$ (resp. $!(U)$) is the type of a *linear channel endpoint* that must be used to input (resp. output) *one* value of type U; $\bullet$ denotes an endpoint that cannot be used for I/O. U is a (possibly recursive) *variant type* where each $\mathtt{l}_i$-labelled element is a *record* with 2 fields: `p` (mapped to a basic value or a linear channel endpoint) and `c` (mapped to a linear endpoint).

▶ **Definition 7.4** ([36]). The *subtyping relation* $\leqslant_\ell$ *between linear types* is coinductively defined by the following rules (where $\leqslant_{\mathbb{B}}$ is a subtyping between basic types):

$$\frac{U \leqslant_\ell U'}{?(U) \leqslant_\ell ?(U')}\ [\leqslant_\ell\text{-In}] \qquad \frac{U' \leqslant_\ell U}{!(U) \leqslant_\ell !(U')}\ [\leqslant_\ell\text{-Out}] \qquad \bullet \leqslant_\ell \bullet\ [\leqslant_\ell\text{-End}] \qquad \frac{V \leqslant_{\mathbb{B}} V'}{V \leqslant_\ell V'}\ [\leqslant_\ell\text{-}\mathbb{B}]$$

$$\frac{\forall i \in I:\ V_i \leqslant_\ell V_i' \quad L_i \leqslant_\ell L_i'}{[\mathtt{l}_i\_\{\mathtt{p}: V_i, \mathtt{c}: L_i\}]_{i\in I} \leqslant_\ell [\mathtt{l}_i\_\{\mathtt{p}: V_i', \mathtt{c}: L_i'\}]_{i\in I\cup J}}\ [\leqslant_\ell\text{-VR}] \qquad \frac{U\{\mu_X.U/_X\} \leqslant_\ell U'}{\mu_X.U \leqslant_\ell U'}\ [\leqslant_\ell\text{-}\mu\text{L}] \qquad \frac{U \leqslant_\ell U'\{\mu_X.U'/_X\}}{U \leqslant_\ell \mu_X.U'}\ [\leqslant_\ell\text{-}\mu\text{R}]$$

The rules in Theorem 7.4 are standard: they include the subtyping for variants and records (rule $[\leqslant_\ell\text{-VR}]$) and left/right recursion ($[\leqslant_\ell\text{-}\mu\text{L}]$/$[\leqslant_\ell\text{-}\mu\text{R}]$). $[\leqslant_\ell\text{-In}]$ and $[\leqslant_\ell\text{-Out}]$ provide respectively the subtyping for linear inputs (*co*variant w.r.t. the subtyping of carried types) and outputs (which is instead *contra*variant): note that they are matched by the variances of `In[·]`/`Out[·]` (Fig. 6, left). By $[\leqslant_\ell\text{-End}]$, $\bullet$ is the only subtype of itself. $[\leqslant_\ell\text{-}\mathbb{B}]$ extends $\leqslant_\ell$ to basic types.

In the linear types world, the *duality* between two channel endpoints is very simple: it holds when they are both $\bullet$, or they are an input and an output carrying the same type.

▶ **Definition 7.5** ([8]). The *dual of* L (written $\overline{L}$) is: $\overline{?(U)} = !(U)$; $\overline{!(U)} = ?(U)$; $\overline{\bullet} = \bullet$.

We now introduce our encoding of session types into linear types. Albeit inspired by [8, 6], it features a different treatment of recursion, allowing us to bridge into Scala types.

▶ **Definition 7.6** (Encoding of session into linear types). Let the *action of a session type* be:

$$\mathrm{act}(\&_{i\in I}?\mathtt{l}_i(T_i).S_i) = ? \qquad \mathrm{act}(\oplus_{i\in I}!\mathtt{l}_i(T_i).S_i) = ! \qquad \mathrm{act}(\mu_X.S) = \mathrm{act}(S)$$

Moreover, let Γ be a partial function from session type variables to linear types. The *encoding of* S *into a linear type w.r.t.* Γ, written $[\![S]\!]_\Gamma$, is defined as:

$$[\![\&_{i\in I}?\mathtt{l}_i(T_i).S_i]\!]_\Gamma = ?([\mathtt{l}_i\_\{\mathtt{p}: [\![T_i]\!], \mathtt{c}: [\![S_i]\!]_\Gamma\}]_{i\in I}) \qquad [\![\oplus_{i\in I}!\mathtt{l}_i(T_i).S_i]\!]_\Gamma = !\left(\left[\mathtt{l}_i\_\left\{\mathtt{p}: [\![T_i]\!], \mathtt{c}: \overline{[\![S_i]\!]_\Gamma}\right\}\right]_{i\in I}\right)$$

$$[\![\&_{i\in I}?\mathtt{l}_i(T_i).S_i]\!]^\mu_\Gamma = [\mathtt{l}_i\_\{\mathtt{p}: [\![T_i]\!], \mathtt{c}: [\![S_i]\!]_\Gamma\}]_{i\in I} \qquad [\![\oplus_{i\in I}!\mathtt{l}_i(T_i).S_i]\!]^\mu_\Gamma = \left[\mathtt{l}_i\_\left\{\mathtt{p}: [\![T_i]\!], \mathtt{c}: \overline{[\![S_i]\!]_\Gamma}\right\}\right]_{i\in I}$$

$$[\![\mu_X.S]\!]_\Gamma = \mathrm{act}(S)\left(\mu_X.[\![S]\!]^\mu_{\Gamma\{\mathrm{act}(S)(X)/_X\}}\right) \qquad [\![\mathbf{end}]\!]_\Gamma = \bullet$$

$$[\![\mu_X.S]\!]^\mu_\Gamma = \mu_X.[\![S]\!]^\mu_{\Gamma\{\mathrm{act}(S)(X)/_X\}} \qquad [\![X]\!]_\Gamma = \Gamma(X) \qquad [\![T]\!]_\Gamma = T\ (\text{if } T \in \mathbb{B})$$

The *encoding of* S *into a linear type* is $[\![S]\!]_\varnothing$, also abbreviated $[\![S]\!]$.

Theorem 7.6 is inductively defined on S. Intuitively, it turns **end** into $\bullet$, and external (resp. internal) choices into linear input (resp. output) types. In the latter case, each choice label becomes a label of the carried variant, its payload is encoded into the `p` field of the corresponding record, and its continuation into the `c` field. Crucially, when encoding an *internal* choice, `c` carries the *dual* of the encoding of the original continuation: this is because, as seen in § 4.3, sending a value requires to allocate a new pair of I/O channel endpoints, keep one of them, and send *the other* (i.e., the dual, by Theorem 7.5) for continuing the session. Recursion is encoded by turning a recursive external (resp. internal) choice into a linear input (resp. output) carrying a *recursive* variant: this "structural shift" is achieved by collecting open recursion variables in Γ, and using the auxiliary encoding $[\![\cdot]\!]^\mu_\Gamma$. E.g., let $S = \mu_X.?\mathtt{A}.X$: $[\![S]\!]_\Gamma$ gives the type $?(\mu_X.U)$, with U obtained by letting $\Gamma' = \Gamma\{^{?(X)}/_X\}$, and $U = [\![?\mathtt{A}.X]\!]^\mu_{\Gamma'} = [\mathtt{A}\_\{\mathtt{p}: \mathsf{Unit}, \mathtt{c}: [\![X]\!]_{\Gamma'}\}] = [\mathtt{A}\_\{\mathtt{p}: \mathsf{Unit}, \mathtt{c}: ?(X)\}]$ (see Theorem 7.11).

Our handling of recursion greatly affects our proofs, and is a main difference between Theorem 7.6 and the encoding in [6]. Despite this, the crucial Theorem 7.2 still holds.

[Encoding preserves duality, subtyping] $\overline{[\![S]\!]} = [\![\overline{S}]\!]$, and $S \leqslant S'$ iff $[\![S]\!] \leqslant_\ell [\![S']\!]$.

7.3 From session types to Scala types

We now present our *encoding of session types into Scala types*. Since Scala has a *nominal* type system but session types are *structural*, our encoding requires a *nominal environment* (Theorem 7.7), giving a distinct class name to each subterm of S.

▶ **Definition 7.7.** A *nominal environment for session types* $\mathcal{N}$ is a partial function from (possibly open) session types to Scala class names. $\mathcal{N}$ *is suitable for* S iff *(i)* $\mathrm{dom}(\mathcal{N})$ contains all subterms of S (except **end**), *(ii)* is injective w.r.t. the internal/external choices in its domain, *(iii)* maps each *singleton* internal/external choice to its label, *(iv)* is *dually closed*, i.e. $\forall S' \in \mathrm{dom}(\mathcal{N}) : \mathcal{N}(S') = \mathcal{N}(\overline{S'})$, and *(v)* if $\mathcal{N}(\mu_X.S')$ is defined, then $\mathcal{N}(\mu_X.S') = \mathcal{N}(X) = \mathcal{N}(S')$.

Our encoding of a session type S into a Scala type is given in Theorem 7.10. It relies on an *intermediate encoding* of S into a linear type L, which is further encoded into Scala classes. Such an intermediate step will allow us to exploit the fact that L is either $\bullet$, or a linear input/output $?(U)/!(U)$, for some (possibly recursive) U. We will see that:

- if L is an input (resp. output), it will result in a `lchannels In[·]` (resp. `Out[·]`) type;
- U also appears in the dual $\overline{L}$ (by Def. 7.5), corresponding to $\overline{S}$ (by § 7.2): it will produce both the type parameter of `In`/`Out` above, and the *CPSP classes* of $S/\overline{S}$.

We first formalise the encoding from linear types to Scala types, in Theorem 7.8 below.

▶ **Definition 7.8.** A *nominal environment for linear types* $\mathcal{M}$ is a partial function from (possibly open) variant types to Scala class names. $\mathcal{M}$ *is suitable for* L iff $\mathrm{dom}(\mathcal{M})$ contains all subterms of L (except $\bullet$), is injective w.r.t. the variants in its domain, maps each *singleton* variant to its label, and if $\mathcal{M}(\mu_X.U)$ is defined, then $\mathcal{M}(\mu_X.U) = \mathcal{M}(X) = \mathcal{M}(U)$. Given $\mathcal{M}$ suitable for L, we define the *encoding of* L *into Scala types w.r.t.* $\mathcal{M}$, written $\langle L \rangle_{\mathcal{M}}$, as:

$$\langle ?(U) \rangle_{\mathcal{M}} = \texttt{In[}\mathcal{M}(U)\texttt{]} \qquad \langle !(U) \rangle_{\mathcal{M}} = \texttt{Out[}\mathcal{M}(U)\texttt{]} \qquad \langle \bullet \rangle_{\mathcal{M}} = \texttt{Unit} \qquad \langle V \rangle_{\mathcal{M}} = V \ (\text{if } V \in \mathbb{B})$$

$$\langle U \rangle_{\mathcal{M}} = \begin{cases} \texttt{case class l (p:}\langle V \rangle_{\mathcal{M}}\texttt{)(val cont:}\langle L \rangle_{\mathcal{M}}\texttt{)} & \text{if } U = [\texttt{l\_}\{\texttt{p} : V, \texttt{c} : L\}] \\ \langle U' \rangle_{\mathcal{M}} \quad \text{if } U' = \mathrm{carr}(V) & \\ \langle U'' \rangle_{\mathcal{M}} \quad \text{if } U'' = \mathrm{carr}(L) & \end{cases}$$

$$\langle U \rangle_{\mathcal{M}} = \begin{cases} \texttt{sealed abstract class } \mathcal{M}(U) & \\ \texttt{case class l}_i\ \texttt{(p:}\langle V_i \rangle_{\mathcal{M}}\texttt{)(val cont:}\langle L_i \rangle_{\mathcal{M}}\texttt{) extends } \mathcal{M}(U) & \Big\}_{i \in I} \text{ if } U = [\texttt{l}_i\texttt{\_}\{\texttt{p} : V_i, \texttt{c} : L_i\}]_{i \in I} \text{ and } |I| > 1 \\ \langle U' \rangle_{\mathcal{M}} \quad \text{if } U' = \mathrm{carr}(V_i) & \\ \langle U'' \rangle_{\mathcal{M}} \quad \text{if } U'' = \mathrm{carr}(L_i) & \end{cases}$$

$$\langle \mu_X.U \rangle_{\mathcal{M}} = \langle U \rangle_{\mathcal{M}} \qquad \langle X \rangle_{\mathcal{M}} = \mathcal{M}(X)$$

The encoding in Theorem 7.8 is inductively defined on L. The first 3 cases turn a top-level $?(\cdot)/!(\cdot)/\bullet$ into a corresponding `In[·]`/`Out[·]`/`Unit` type in Scala, and the 4th case keeps basic types unaltered; note that when encoding $?(U)$ (resp. $!(U)$), the type parameter of the resulting `In[·]` (resp. `Out[·]`) is the Scala class name that $\mathcal{M}$ maps to U. The remaining cases of Theorem 7.8 show how U originates the session protocol classes. Singleton variants are turned into `case class`es, while non-singleton variants are turned into `sealed abstract class`es (with a name given by $\mathcal{M}$), extended by one `case class` per label. Note that if the `p` field of a variant consists in some linear type $?(U')/!(U')$, the CPSP classes of U' are generated as well – and similarly for the `c` field. A recursive term $\mu_X.U$ is handled by noticing that, by Theorem 7.7, $\mathcal{M}(\mu_X.U) = \mathcal{M}(X) = \mathcal{M}(U)$: hence, X is encoded as $\mathcal{M}(X) = \mathcal{M}(\mu_X.U)$.

The last ingredient for our encoding is a way to turn a nominal environment for a session type (Theorem 7.7) into one for a linear type (Theorem 7.8): this is formalised below.

▶ **Definition 7.9.** We say that S *maps* S' *to* U' (in symbols, $S \vdash S' \mapsto U'$) iff, for some Γ, the computation of $[\![S]\!]$ involves either (a) an instance of $[\![S']\!]_\Gamma$ returning $?(U')$ or $!(U')$, or (b) an instance of $[\![S']\!]^\mu_\Gamma$ returning U'. If $\mathcal{N}$ is suitable for S, the *linear encoding of* $\mathcal{N}$ *(w.r.t.* S*)* is a nominal environment for linear types denoted with $[\![\mathcal{N}]\!]_S$, such that:

$$[\![\mathcal{N}]\!]_S(U) = \mathtt{A} \quad \text{iff} \quad \exists S' \colon S \vdash S' \mapsto U \text{ and } \mathcal{N}(S') = \mathtt{A}$$

Intuitively, Theorem 7.9 says that if $\mathcal{N}$ maps an internal/external choice S' to some class name $\mathtt{A}$, then $[\![\mathcal{N}]\!]_S$ maps the variant obtained from the encoding of S' to the same $\mathtt{A}$.

We are now ready to define our encoding of session types into Scala types.

▶ **Definition 7.10.** Given $\mathcal{N}$ suitable for S, we define the *encoding of* S *into a Scala type* as $\langle\!\langle S \rangle\!\rangle_{\mathcal{N}} = \langle [\![S]\!] \rangle_{[\![\mathcal{N}]\!]_S}$, and the *protocol classes of* S as: $\mathrm{prot}\langle\!\langle S \rangle\!\rangle_{\mathcal{N}} = \langle \mathrm{carr}([\![S]\!]) \rangle_{[\![\mathcal{N}]\!]_S}$.

Theorem 7.10 gives us two pieces of information: $\langle\!\langle S \rangle\!\rangle_{\mathcal{N}}$ is the type `In[·]`/`Out[·]`/`Unit` on which a Scala program can communicate according to S, and $\mathrm{prot}\langle\!\langle S \rangle\!\rangle_{\mathcal{N}}$ gives the definitions of all necessary CPSP classes. Technically, S and $\mathcal{N}$ are first linearly encoded (via Definitions 7.6 and 7.9); then, the result is further encoded into Scala types (via Theorem 7.8).

▶ **Example 7.11.** The linear encoding of the greeting session type S_h in § 2 is:

$$[\![S_h]\!] = !(U_h) \qquad \text{where } U_h = \mu_X.\left[\begin{array}{l} \mathtt{Greet\_}\left\{\mathtt{p}:\mathsf{String}, \mathtt{c}:!\left(\left[\begin{array}{l}\mathtt{Hello\_}\{\mathtt{p}:\mathsf{String}, \mathtt{c}:!(X)\},\\ \mathtt{GoodNight\_}\{\mathtt{p}:\mathsf{String}, \mathtt{c}:\bullet\}\end{array}\right]\right)\right\}, \\ \mathtt{Quit\_}\{\mathtt{p}:\mathsf{Unit}, \mathtt{c}:\bullet\}\end{array}\right]$$

Let us now define $\mathcal{N}$, as described in Theorem 4.2, making it suitable for S_h (as per Theorem 7.7):

$$\mathcal{N}\left(\begin{array}{l}!\mathtt{Greet}(\mathsf{String}).\left(\begin{array}{l}?\mathtt{Hello}(\mathsf{String}).X\\ \&\ ?\mathtt{Bye}(\mathsf{String}).\mathbf{end}\end{array}\right)\\ \oplus\ !\mathtt{Quit}(\mathsf{Unit})\end{array}\right) = \mathtt{Start} \qquad \begin{array}{l}\mathcal{N}(S_h) = \mathtt{Start}\\ \mathcal{N}(X) = \mathtt{Start}\end{array} \qquad \mathcal{N}\left(\begin{array}{l}?\mathtt{Hello}(\mathsf{String}).X\\ \&\ ?\mathtt{Bye}(\mathsf{String}).\mathbf{end}\end{array}\right) = \mathtt{Greeting}$$

Now, we can verify that the following mappings hold:

$$S_h \vdash \left(\begin{array}{l}!\mathtt{Greet}(\mathsf{String}).\left(\begin{array}{l}?\mathtt{Hello}(\mathsf{String}).X\\ \&\ ?\mathtt{Bye}(\mathsf{String}).\mathbf{end}\end{array}\right)\\ \oplus\ !\mathtt{Quit}(\mathsf{Unit})\end{array}\right) \mapsto \left[\begin{array}{l} \mathtt{Greet\_}\left\{\mathtt{p}:\mathsf{String}, \mathtt{c}:!\left(\left[\begin{array}{l}\mathtt{Hello\_}\{\mathtt{p}:\mathsf{String}, \mathtt{c}:!(X)\},\\ \mathtt{Bye\_}\{\mathtt{p}:\mathsf{String}, \mathtt{c}:\bullet\}\end{array}\right]\right)\right\}, \\ \mathtt{Quit\_}\{\mathtt{p}:\mathsf{Unit}, \mathtt{c}:\bullet\}\end{array}\right]$$

$$S_h \vdash S_h \mapsto U_h \qquad S_h \vdash X \mapsto X \qquad S_h \vdash \left(\begin{array}{l}?\mathtt{Hello}(\mathsf{String}).X\\ \&\ ?\mathtt{Bye}(\mathsf{String}).\mathbf{end}\end{array}\right) \mapsto \left[\begin{array}{l}\mathtt{Hello\_}\{\mathtt{p}:\mathsf{String}, \mathtt{c}:!(X)\},\\ \mathtt{Bye\_}\{\mathtt{p}:\mathsf{String}, \mathtt{c}:\bullet\}\end{array}\right]$$

Hence, by Theorem 7.9, $[\![\mathcal{N}]\!]_{S_h}$ maps the first, second and third (recursive) variant types above to `Start`, and the last one to `Greeting`. The encoding $\langle\!\langle S_h \rangle\!\rangle_{\mathcal{N}} = \langle [\![S_h]\!] \rangle_{[\![\mathcal{N}]\!]_{S_h}}$ is `Out[Start]`, while $\mathrm{prot}\langle\!\langle S_h \rangle\!\rangle_{\mathcal{N}} = \langle \mathrm{carr}([\![S_h]\!]) \rangle_{[\![\mathcal{N}]\!]_{S_h}}$ gives the Scala protocol classes seen in Theorem 4.2.

We conclude with two results at the roots of our session-based development approach (§ 4.2). Letthe *dual of a Scala type* be $\overline{\mathtt{In[A]}} = \mathtt{Out[A]}$, $\overline{\mathtt{Out[A]}} = \mathtt{In[A]}$, and $\overline{\mathtt{Unit}} = \mathtt{Unit}$.

For all S, $\langle\!\langle \overline{S} \rangle\!\rangle_{\mathcal{N}} = \overline{\langle\!\langle S \rangle\!\rangle_{\mathcal{N}}}$ and $\mathrm{prot}\langle\!\langle S \rangle\!\rangle_{\mathcal{N}} = \mathrm{prot}\langle\!\langle \overline{S} \rangle\!\rangle_{\mathcal{N}}$.

§ 7.3 says that a session type and its dual are encoded as *dual Scala types*, and dual session types have *the same protocol classes*: this justifies steps **D1**–**D3** in § 4.2.

Finally, let $<:$ be the Scala subtyping (the full definition is available in the on-line technical report, see § 1). Suppose that we encode a session type S, getting $\mathtt{B}$, and write a program using $\mathtt{A}$ such that $\mathtt{A} <: \mathtt{B}$ or $\mathtt{B} <: \mathtt{A}$: by § 7.3, this is sound. For all $\mathtt{A}, S, \mathcal{N}$, $\mathtt{A} <: \langle\!\langle S \rangle\!\rangle_{\mathcal{N}}$ implies one of the following:

(a1) $S = \mathbf{end}$, and: $\mathtt{A} <: \mathtt{Unit}$ and $\forall \mathtt{B}: \mathtt{A} \notin \{\mathtt{In[B]}, \mathtt{Out[B]}\}$;
(a2) $\mathrm{act}(S) = ?$, and: $\mathtt{A} <: \mathtt{Null}$ or $\exists \mathtt{B}: \mathtt{A} = \mathtt{In[B]}$ and ($\mathtt{Null} \not<: \mathtt{B}$ implies $\exists S', \mathcal{N}' : \mathtt{A} = \langle\!\langle S' \rangle\!\rangle_{\mathcal{N}'}$ and $S' \leqslant S$);
(a3) $\mathrm{act}(S) = !$, and: $\mathtt{A} <: \mathtt{Null}$ or $\exists \mathtt{B}: \mathtt{A} = \mathtt{Out[B]}$ and ($\mathtt{B} \not<: \mathtt{AnyRef}$ implies $\mathtt{A} = \langle\!\langle S \rangle\!\rangle_{\mathcal{N}}$).

Moreover, for all $\mathtt{A}, S, \mathcal{N}$, $\langle\!\langle S \rangle\!\rangle_{\mathcal{N}} <: \mathtt{A}$ implies one of the following:

(b1) $S = \mathbf{end}$, and: `Unit <: A` and $\forall$`B: A` $\notin \{$`In[B]`, `Out[B]`$\}$;
(b2) $\mathrm{act}(S) = ?$, and: `AnyRef <: A` or $\exists$`B: A = In[B]` and (`B` $\not\geqslant$`: AnyRef` implies `A` $= \langle\!\langle S \rangle\!\rangle_{\mathcal{N}}$);
(b3) $\mathrm{act}(S) = !$, and: `AnyRef <: A` or $\exists$`B: A = Out[B]` and (`Null` $\not\geqslant$`: B` implies $\exists S', \mathcal{N}'$: `A` $= \langle\!\langle S' \rangle\!\rangle_{\mathcal{N}'}$ and $S \leqslant S'$).

Roughly, § 7.3 says that Scala subtyping reflects session subtyping, thus preserving its safety/exhaustiveness guarantees (**S1** and **S2** in § 2.1). When **end** is encoded, items a1/b1 say that its Scala sub/super-types cannot be `In`/`Out`, i.e. their instances do not allow I/O. For item a2, consider Theorem 4.3: we have `In[Full] <: In[WaitingRoom]`, reflecting the fact that `?Full` $\leqslant S_{\mathrm{cstm}}$ (by [$\leqslant$-Ext]). For item b3, consider Theorem 4.2: we have `Out[Start] <: Out[Quit]`, reflecting the fact that $S_{\mathrm{h}} \leqslant$ `!Quit` (by [$\leqslant$-μL] and [$\leqslant$-Int]). § 7.3 also says that `<:` is stricter than $\leqslant$ – e.g., by item a3, the Scala encoding of an internal choice has no subtypes, and by item b2, an external choice has no supertypes. However, Scala allows for sub/super-types that *do not correspond to any session type*: besides the unavoidable `Null` cases (items a2, a3, b3), it is possible e.g. to write a method `f` with a parameter of type `In[Any]` (b2), or `In[Nothing]` (a2), or `Out[Any]` (a3), or `Out[Nothing]` (b3). This does not compromise safety/exhaustiveness, either: `In[Any]` makes `f` accept any message, `Out[Nothing]` forbids `f` to send, while `In[Nothing]`/`Out[Any]` are subtypes of all `In`/`Out` types – thus making `f` non-applicable to any channel endpoint obtained by encoding a session type. Notably, this holds by co/contra-variance of `In[+A]`/`Out[-A]` (Fig. 6, left).

8 Related work

Session types and their implementation. Session types were introduced by Honda *et al.* in [18, 39, 19], as a typing discipline for a variant of the π-calculus (called session-π in § 2). They have been studied and developed in multiple directions during the following decades, notably addressing *multiparty* interactions [20] and logical interpretations [5, 43]. The encoding of session types in linear π-calculus types has been studied in [9, 8, 6, 7]; our work is mainly based on [8], but our treatment of recursion is novel (see § 7.2).

Session types have been mostly implemented on dedicated programming languages with the advanced type-level features outlined in § 2 [14, 43, 11, 40, 3]. [32, 34] aim at an integration with Haskell, using monads to enforce linearity (at the price of a restrictive and rather complicated API). [25] adapts [34] to Rust, exploiting its *affine types*, but showing limitations to *binary* internal/external choices. [23, 37, 38] are based on a Java language extension and runtime with session-type-inspired primitives for I/O and branching. [22] integrates session types in Java via automatic generation of classes representing session-typed channel endpoints, with run-time linearity checks. The main differences w.r.t. our work are that [22] is closer to session-π, is based on the Scribble tool [44], supports *multiparty* sessions, and generates classes which represent *both* a channel endpoint *and* its protocol; hence, in the binary setting, each endpoint has *its own* hierarchy of generated classes that is different (but "dual") w.r.t. the other endpoint. Instead, our I/O endpoints are closer to linear types for the π-calculus [36]: they take the protocol as a type parameter, from a set of CPSP classes which is common between the two endpoints. Other differences are mostly due to the Java type system, which e.g. does not support `case class`es (complicating exhaustiveness checks) nor declaration-site variance (complicating the handling of I/O co/contra-variance).

The work closer to ours is [33]: it presents an encoding of session types in a ML-like language, and an OCaml library reminiscent of `lchannels`. We share several ideas and features, including the theoretical basis of [8]. The differences are at technical and API design levels, due to different languages and goals (type inference vs. CPSP extraction);given the wide adoption of Scala, we focus on practical validation with use cases and benchmarks.

Strong typing guarantees for concurrent applications have been a longstanding goal for

the Scala and Akka communities. In the actor realm, Akka Typed (§ 1) is remarkably close to [17]: both propose `ActorRef[A]`-typed actor references. We drew inspiration from them and CPSPs, merging the theoretical basis of [8]. Some (non-linear) channel APIs have been tentatively introduced in Akka, e.g. *channels* (Akka 1.2) and macro-based *typed channels* (Akka 2.1); however, they were later deprecated, mainly due to design and maintainability issues [26]. `lchannels` is based on a clear and well-established theory, adapted to the Scala setting: thus, the implementation is fairly simple and maintainable, not requiring macros.

9 Conclusions

We showed how *session programming* can be carried over in Scala, by representing protocols as types that the compiler can check. We based our approach on a *lightweight* integration of *session types*, based on *CPSP classes* and the `lchannels` library. We showed that our approach supports *local* and *distributed* interaction, has a formal basis (the *encoding of session types into linear I/O types*), and attested its viability with use cases and benchmarks.

We plan to extend our approach to *multiparty* session types (MPSTs), by extracting CPSP classes from a *global type* [20], rather than addressing multiple binary session separately (as in Theorem 4.3 and § 6.1). Just as binary session typing guarantees safe and deadlock free interaction for *two* parties involved in one session (§ 2.2), MPSTs extend such a guarantee to two *or more* parties; the main challenge is that encoding MPSTs into Scala types might be complex, and require a tool akin to [22].

The Scala landscape is fast-moving, and recent developments may influence the evolution of our work. [41] introduces customisable effect for Scala: by extending the `lchannels` I/O operations with an effect, we could obtain stronger linearity guarantees – e.g., ensuring that a program does not "forget" a session (§ 6.1.3). [15] studies capabilities for borrowing object references: they could ensure that a channel endpoint is never used if sent (§ 3.1). Similar guarantees could be achieved by examining the program call graph [1]. Recent results on Scala's type system (e.g. on path-dependant and structural types [2, 35]) might improve our encoding, removing the limitation on the uniqueness of choice labels (Remark 7.1).

We will further extend and optimise `lchannels` and its API: many improvements are possible, and the transport abstraction allows to easily compare different implementations, under different settings and uses. We also plan to extend our approach to other languages: one candidate is C#, due to its support for first-class functions and declaration-site variance.

Towards session types for Akka Typed (and other frameworks). This work focuses on `lchannels`, but our approach can be generalised to other communication frameworks. One possible way is abstracting under the `In[·]`/`Out[·]` API, as in § 5; another way is *directly using the I/O endpoints offered by other frameworks*. Consider e.g. Akka Typed: we can adapt CPSP extraction (Theorem 7.8) to yield `ActorRef[A]` types instead of `Out[A]`, obtaining CPSP classes similar to those in Fig. 2. Remarkably, `Out[A]` and `ActorRef[A]` are both contravariant w.r.t. `A`, and enjoy similar subtyping properties (§ 7.3). However:

- **(i)** Akka Typed does not offer an *input* endpoint similar to `In[·]`. Hence, session types whose CPSPs carry *input* endpoints (e.g., Theorem 4.1, or S_{rctl} in § 6.1.1) must be adapted (i.e., sequences of two outputs or two inputs must be replaced with input-output alternations);
- **(ii)** instances of `ActorRef[A]` raise no errors when used multiple times for sending messages;
- **(iii)** to produce and send a continuation `ActorRef[A]`, it is customary to cede the control to another actor (possibly a new one, as in Fig. 3); `lchannels`, instead, encourages the creation and use of I/O endpoints along a single thread, in a simple sequential style.

Item 1 is a minor issue; 2 could be addressed, taking inspiration from the session/linear types theory, by distinguishing *unrestricted* [42] `ActorRef`s (allowing 0 or more outputs of the same type) from *linear* `ActorRef`s – with the former usable as the latter, but not *vice versa*. Item 3 marks a crucial difference between *reactive, actor-based concurrent programming* (where the protocol flow is decomposed into multiple input-driven handlers), and *thread-based* programming. We plan to study the formal foundations for applying "session types as CPSPs" in the *reactive* setting, and their feasibility w.r.t. software industry practices.

Acknowledgements. Thanks to Roland Kuhn, Julien Lange and the anonymous reviewers for their helpful remarks on earlier versions of this paper. Thanks to Julien Lange and Nicholas Ng for their feedback during artifact testing, and to the anonymous artifact reviewers for their detailed remarks and suggestions.

References

1 Karim Ali, Marianna Rapoport, Ondřej Lhoták, Julian Dolby, and Frank Tip. Constructing call graphs of Scala programs. In *ECOOP*, 2014. doi:10.1007/978-3-662-44202-9_3.

2 Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *OOPSLA*, 2014. doi:10.1145/2660193.2660216.

3 Stephanie Balzer and Frank Pfenning. Objects as session-typed processes. In *AGERE!*, 2015. doi:10.1145/2824815.2824817.

4 Hendrik Pieter Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North Holland, 1985.

5 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, 2010. doi:10.1007/978-3-642-15375-4_16.

6 Ornela Dardha. Recursive session types revisited. In *BEAT*, 2014. doi:10.4204/EPTCS.162.4.

7 Ornela Dardha. *Type Systems for Distributed Programs: Components and Sessions*. Phd thesis, Università degli studi di Bologna, May 2014.

8 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP*, 2012. doi:10.1145/2370776.2370794.

9 Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In *CONCUR*, 2011.

10 Edsger W Dijkstra. *Cooperating sequential processes*. Springer, 1965.

11 Juliana Franco and Vasco Thudichum Vasconcelos. A concurrent programming language with refined session types. In *SEFM*, 2013. doi:10.1007/978-3-319-05032-4_2.

12 Simon Gay and Malcolm Hole. Types and subtypes for client-server interactions. In *ESOP*. Springer Berlin Heidelberg, 1999. doi:10.1007/3-540-49099-X_6.

13 Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 2005. doi:10.1007/s00236-005-0177-z.

14 Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1), January 2010. doi:10.1017/S0956796809990268.

15 Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *ECOOP*, 2010.

16 Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Futures and Promises. URL: http://docs.scala-lang.org/overviews/core/futures.html.

17 Jiansen He, Philip Wadler, and Philip Trinder. Typecasting actors: From Akka to TAkka. In *SCALA'14*, 2014. doi:10.1145/2637647.2637651.

18 Kohei Honda. Types for dyadic interaction. In *CONCUR*, 1993. doi:10.1007/3-540-57208-2_35.

19 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, 1998. doi:10.1007/BFb0053567.

20 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, 2008. doi:10.1145/1328438.1328472.

21 Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in Java. In *ECOOP*, 2010. doi:10.1007/978-3-642-14107-2_16.

22 Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In *FASE*, 2016.

23 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *ECOOP*, 2008. doi:10.1007/978-3-540-70592-5_22.

24 Shams M. Imam and Vivek Sarkar. Savina – an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE!, 2014. doi:10.1145/2687357.2687368.

25 Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP, 2015. doi:10.1145/2808098.2808100.

26 Roland Kuhn. Project Gålbma, actors vs types, 2015. Slides (available on slideshare.net).

27 Lightbend, Inc. Actor paths, 2016. http://doc.akka.io/.../addressing.html.

28 Lightbend, Inc. The Akka toolkit and runtime, 2016. URL: http://akka.io/.

29 Lightbend, Inc. Akka Typed, 2016. http://doc.akka.io/.../typed.html.

30 Lightbend, Inc. The Scala IDE, 2016. URL: http://scala-ide.org/.

31 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Inf. & Comput.*, 1992. doi:10.1016/0890-5401(92)90008-4.

32 Matthias Neubauer and Peter Thiemann. An implementation of session types. In *PADL*, 2004. doi:10.1007/978-3-540-24836-1_5.

33 Luca Padovani. A Simple Library Implementation of Binary Sessions. hal:01216310, 2015.

34 Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Haskell*, 2008. doi:10.1145/1411286.1411290.

35 Tiark Rompf and Nada Amin. From F to DOT: Type soundness proofs with definitional interpreters. Technical report, Purdue University and EPFL, 2015. Unpublished. http://arxiv.org/abs/1510.05216. arXiv:1510.05216.

36 Davide Sangiorgi and David Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2003.

37 K. C. Sivaramakrishnan, Karthik Nagaraj, Lukasz Ziarek, and Patrick Eugster. Efficient session type guided distributed interaction. In *COORDINATION*, 2010. doi:10.1007/978-3-642-13414-2_11.

38 K. C. Sivaramakrishnan, Mohammad Qudeisat, Lukasz Ziarek, Karthik Nagaraj, and Patrick Eugster. Efficient sessions. *Sci. Comput. Program.*, 78(2), 2013. doi:10.1016/j.scico.2012.03.004.

39 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE*, 1994. doi:10.1007/3-540-58184-7_118.

40 Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: a monadic integration. In *ESOP*, 2013. doi:10.1007/978-3-642-37036-6_20.

41 Matías Toro and Éric Tanter. Customizable gradual polymorphic effects for Scala. In *OOPSLA*, 2015. doi:10.1145/2814270.2814315.

42 Vasco T. Vasconcelos. Fundamentals of session types. *Inf. & Comput.*, 217, 2012. doi:10.1016/j.ic.2012.05.002.

43 Philip Wadler. Propositions as sessions. In *ICFP*, 2012. doi:10.1145/2364527.2364568.

44 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In *TGC*, 2013. doi:10.1007/978-3-319-05119-2_3.

Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java

Johannes Späth[1], Lisa Nguyen Quang Do[*2], Karim Ali[3], and Eric Bodden[†4]

1 **Fraunhofer SIT**
johannes.spaeth@sit.fraunhofer.de
2 **Universität Paderborn and Fraunhofer IEM**
lisa.nguyen@iem.fraunhofer.de
3 **Technische Universität Darmstadt**
karim.ali@cased.de
4 **Heinz Nixdorf Institut, Universität Paderborn and Fraunhofer IEM**
eric.bodden@uni-padeborn.de

Abstract

Many current program analyses require highly precise pointer information about small, targeted parts of a given program. This motivates the need for demand-driven pointer analyses that compute information only where required. Pointer analyses generally compute points-to sets of program variables or answer boolean alias queries. However, many client analyses require richer pointer information. For example, taint and typestate analyses often need to know the set of *all aliases* of a given variable *under a certain calling context*. With most current pointer analyses, clients must compute such information through repeated points-to or alias queries, increasing complexity and computation time for them.

This paper presents Boomerang, a demand-driven, flow-, field-, and context-sensitive pointer analysis for Java programs. Boomerang computes rich results that include both the possible allocation sites of a given pointer (points-to information) and all pointers that can point to those allocation sites (alias information). For increased precision and scalability, clients can query Boomerang with respect to particular calling contexts of interest.

Our experiments show that Boomerang is more precise than existing demand-driven pointer analyses. Additionally, using Boomerang, the taint analysis FlowDroid issues up to 29.4x fewer pointer queries compared to using other pointer analyses that return simpler pointer information. Furthermore, the search space of Boomerang can be significantly reduced by requesting calling contexts from the client analysis.

1998 ACM Subject Classification F.3.2 – Logics and Meanings of Programs – Semantics of Programming Languages—Program Analysis

Keywords and phrases Demand-Driven; Static Analysis; IFDS; Aliasing; Points-to Analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.22

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.2.1.12

[*] Research was conducted while being at Fraunhofer SIT
[†] Research was conducted while being at Technische Universität Darmstadt and Fraunhofer SIT

30th European Conference on Object-Oriented Programming (ECOOP 2016).
Editors: Shriram Krishnamurthi and Benjamin S. Lerner; Article No. 22; pp. 22:1–22:26

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
1 context1(){
2   A d = new A();
3   A e = new A();
4   String f = source();
5   foo(d,e,f);
6 }
7 context2(){
8   A x = new A();
9   A y = x;
10  String z = noSource();
11  foo(x,y,z);
12 }
13 foo(A a, A b, String s){
14   a.f = s;
15   sink(b.f);
16 }
```

Figure 1 An example program that illustrates the problem of finding all aliases under a specific calling context.

1 Introduction

Static analysis nowadays is often used for vulnerability detection, finding bugs, and program understanding tools. This setting typically requires the underlying analyses —including pointer analyses— to compute information on-demand (i.e., only for the portions of the program that are of specific interest to the client that is querying them). In the case of pointer analyses, the different clients are interested in different types of pointer information [1, 28]. For example, a call graph analysis [26] approximates the runtime types of call receivers using points-to sets that contain all possible allocation sites of an object, and their runtime types. On the other hand, race-detection algorithms [18], typically need to know whether two given program variables may alias (i.e., point to the same memory location). Traditionally, pointer analyses can be categorized into either *points-to analyses* that compute points-to sets or *alias analyses* that give a boolean answer to an alias query. As we show in this work, computing only one kind of pointer information is ill-suited when supporting some client analyses. In particular, taint analysis [1] and typestate analysis [3, 16] have to post-process both types of pointer information, as they are interested in *all aliases* of a given variable.

Figure 1 shows an example program that illustrates this problem. In the case of a taint analysis starting at line 4, the taint flows into method `foo` and reaches the field write statement at line 14. To process this statement, the taint analysis must taint `a.f` and all of its aliases, which requires finding all aliases of `a`. Traditional alias analyses do not return this information directly, which is why the taint analysis has to query the alias analysis repeatedly: it must iterate over all existing local variables and determine if they alias with `a`. Additionally, a precise taint analysis should be able to restrict its pointer queries to specific calling contexts. In the example, a false positive is reported at line 15, if the taint analysis merges `context1`, where a taint is flowing but no aliases are created, and `context2`, where `a` and `b` effectively alias (due to the assignment at line 9) but no taint is issued.

As shown in our experiments, repeated queries impede performance and context-insensitive queries impede precision. Ideally, a taint analysis should be able to issue a unique pointer query at line 14 for the variable `a` *under a specific, client-defined calling context* and directly obtain *all possible aliases* of `a`. The result of such a query would be the alias set `{a}` under `context1()` (or `{a,b}` under `context2()` if that query is issued instead). Due to the lack of such sophisticated pointer analyses, some client analyses are tightly integrated with custom, client-specific pointer analyses [1, 28]. This drastically complicates their coordination and prevent the reuse of these custom pointer analysis.

In this paper, we present Boomerang, the first reusable on-demand pointer analysis that, for a given variable, computes both *points-to* information and *all alias* information. Boomerang is flow-, field-, and context-sensitive. For each query, Boomerang performs a backward analysis to collect points-to information and then proceeds with a forward analysis to determine all access graphs [13] that point to the discovered allocation sites. For

the program in Figure 1, if a client queries Boomerang at line 14 for variable `a` under `context1()`, Boomerang first searches backwards for all related allocation sites, discovering the allocation at line 2. The forward analysis then computes that, in `foo` under `context1()`, only variable `a` points to that allocation site.

To increase efficiency, Boomerang is built on top of the IFDS framework for Inter-procedural Finite Distributive Subset problems [19]. This design choice allows Boomerang to store and reuse the results of intra-procedural computations in the form of highly reusable, extremely fine-grained summaries that reason about individual access graphs. Since IFDS can only solve subset problems with distributive flow functions—and pointer analysis is known to be a non-distributive problem—Boomerang uses IFDS only for sub-tasks that can be expressed in a distributive way. Non-distributive fragments of the analysis are handled through additional iterations at *points of indirection*, typically accesses to the heap and at context switches.

We have evaluated the precision and recall of Boomerang using PointerBench, a novel micro-benchmark suite created to evaluate pointer analyses by indicating aliases/non-aliases pairs, as well as points-to sets. In comparison to the demand-driven points-to analysis by Sridharan and Bodík (SB) [23] and the demand-driven alias analysis by Yan et al. (DA) [30], Boomerang reports fewer false-positives with respect to alias pairs and allocation sites. We also compare the effect of using Boomerang, SB, and DA in the client taint analysis FlowDroid. On average, FlowDroid issues 19.9x and 29.4x more queries when using DA and SB, respectively, compared to using Boomerang. Moreover, since SB and DA are less precise than Boomerang, they cause FlowDroid to report more false positives.

Boomerang applies the notion of context-resolution by querying the client analysis for calling contexts as it computes pointer information. Unlike previous approaches that visit all contexts, Boomerang only considers contexts of interest to the client. This improves both the precision of the client (as shown in Figure 1 with the false positive at line 15) and the overall scalability (by propagating along less calling contexts). To evaluate the efficiency of context-resolution, we measure the percentage of contexts filtered out by FlowDroid, a taint analysis client, when using Boomerang. Our experiments show that FlowDroid filters out up to 96% of the calling contexts, which represents a significant computational saving for Boomerang.

To summarize, this paper makes the following contributions:

- We present Boomerang, a demand-driven flow- and context-sensitive pointer analysis that provides both points-to and all alias information. To our knowledge, this is the first analysis that directly provides both types of pointer information.
- We evaluate the precision and recall of Boomerang compared to the state-of-the-art demand-driven pointer analyses on PointerBench, a micro-benchmark suite for pointer analyses.
- We compare the effect of using Boomerang to using other demand-driven pointer analyses when integrated into a taint analysis client.

2 Background

IFDS is a framework for solving inter-procedural finite distributive subset problems [19]. If an analysis can be expressed as such, the framework reduces it to a reachability problem on an *exploded supergraph*, a directed graph representing the analyzed program. Figure 2 shows the exploded supergraph of an example Java program. Throughout this section, we

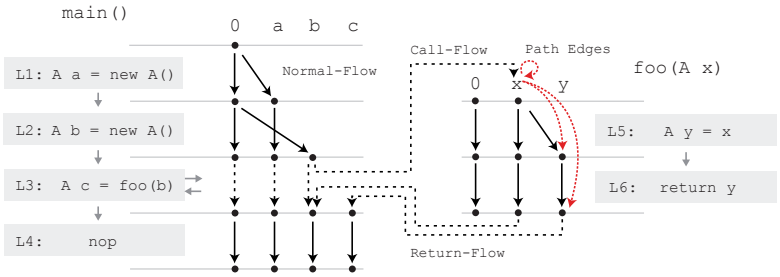


Figure 2 Creation of the exploded supergraph and summary edges in IFDS.

will use this example to explain how a whole-program points-to analysis can be computed using IFDS. For the sake of simplicity, we omit field accesses.

Data-Flow Domain

An IFDS-based analysis defines a *data-flow domain* D, composed of the *data-flow facts* whose validity is inferred by the analysis at each statement of the analyzed program. In the case of pointer analyses, facts typically represent references to objects in the analyzed program, but one can also encode richer information like typestate information [16]. In Figure 2, the fact `0` represents the tautological fact that always holds. It is the root of the exploded super graph. Before statement **L3**, facts `a` and `b` hold because they are reachable from `0`—both variables may point to objects created earlier in the program. Fact `c`, on the other hand, only holds after **L3** since it has not been defined before.

Flow Functions

In addition to the domain D, the analysis has to define *flow functions*. An IFDS flow function is a mapping from D to 2^D. It transfers a single fact to a (possibly empty) set of facts at the successors of a given statement. The composition of the flow functions incrementally creates the exploded supergraph. Flow functions can be of one of four types:

- **Normal flow functions** are applied to every non-call statement. For example, the flow function at statement **L2** transfers `a` to `a` because `a` continues to point to the same object. At the same statement, the flow function applied to `0` generates `b`, modeling a new points-to relationship for `b`.
- **Call flow functions** occur at invoke statements. They map facts from a caller's scope to the callee's scope such as replacing actual arguments by formal parameters.
- **Return flow functions** are the inverse functions of the call-flow functions, mapping the facts back to the scope of the caller.
- **Call-to-return flow functions** locally propagate the information that is not affected by a call (represented by the straight dashed edges in method `main` in Figure 2).

At control-flow merge points, IFDS merges the result of all incoming flow functions using the set union: a node is reachable if it is reachable along *any* path through the exploded supergraph. IFDS assumes that the flow functions are distributive with respect to set union (i.e., $f(A \cup B) = f(A) \cup f(B)$). Distributivity is a key property of the framework as it enables effective reuse of fine-grained per-fact procedure-summaries [19].

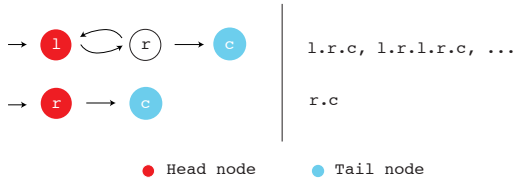


Figure 3 Access graph and access path correlation.

Path Edges

In the exploded supergraph, an existing fact holds (i.e., is reachable) if its predecessors are. This means that the solution to the IFDS problem is given by the facts reachable from the root fact `0`. To compute reachability, IFDS incrementally constructs *path edges* within the scope of methods. A path edge has the form $\langle s, d_1 \rangle \rightarrow \langle t, d_2 \rangle$, where s and t are statements and $d_1, d_2 \in D$. All path edges are intra-procedural and start from a method's entry point. The existence of a path edge summarizes a node's reachability in the exploded supergraph in relative terms: d_2 is reachable at t in every calling context in which d_1 is reachable at the method entry point s.

Figure 2 illustrates the construction of path edges in method `foo` (represented by red dotted arrows). First, IFDS creates a path edge from the starting point of the method, **SP**: $\langle \mathbf{SP}, x \rangle \rightarrow \langle \mathbf{SP}, x \rangle$. This is then extended to $\langle \mathbf{SP}, x \rangle \rightarrow \langle \mathbf{L5}, y \rangle$ and finally to $\langle \mathbf{SP}, x \rangle \rightarrow \langle \mathbf{L6}, y \rangle$. Unlike IFDS, path edges in Boomerang do not necessarily start from a method's entry point. We allow path edges to also start at allocation sites and call sites. This way, Boomerang can encode the reachability of allocation sites into the path edges. We discuss this approach in more detail in Section 3.4.

The path edges ending at a method's exit point are turned into *method summaries*, which avoids any re-evaluation of the same method for the same incoming facts. Contexts are thus quantified-over symbolically: a summary edge is applicable to every context in which the edge's source fact holds.

Access Graphs

To support field-sensitivity (i.e., distinguishing fields not only by their name, but also by their base object), Boomerang models data-flow facts using *access graphs* [13]. An access graph is defined as follows: let $\mathcal{L}$ be the set of all variables of the analyzed program and $\mathcal{F}$ the set of all existing fields of all classes within the program. An *access graph* has two main elements: (1) the *base*, a local variable $l \in \mathcal{L}$, and (2) the *field graph*, a directed graph of nodes from $\mathcal{F}$ that represent field accesses. If the field graph is empty, the data-flow fact represents a plain local variable. Otherwise, the field graph determines through which field accesses an object of interest is reachable. If the field graph is not empty, two nodes (possibly the same) are marked as the *head* and the *tail*. In addition, for each node n within the graph, there exists a path from the head to the tail passing through n. A field graph can then be uniquely identified by its edge set, its head, and its tail. We use $[l, s, E, t] \in \mathcal{L} \times \mathcal{F} \times 2^{(\mathcal{F} \times \mathcal{F})} \times \mathcal{F}$ to denote an access graph with local variable l, head node s, tail t, and edge set of the field graph E. We abbreviate the domain of all access graphs as $\mathcal{APG}$. Throughout the rest of the paper, we overset variables with $\tilde{\cdot}$ as a shorthand notation for an access graph.

An access graph corresponds to (infinitely) many *access paths*: all paths obtained by traversing the field graph from the head to the tail. Figure 3 shows two access graphs and their corresponding sets of access paths.

We use operations on access graphs presented in earlier work [13]. These operations capture the transformations of the access graphs mostly at field read and write statements, where fields are consumed or added to the graph.

- **Remove Head:** for each successor field s' of the head s, we derive a new access graph with s' as the head. The appropriate edge is removed, if s is not part of a loop. This operation assumes a non-empty field graph.

$$\ominus[y, s, E, t] := \begin{cases} \{[y, s', E, t]\}, & \text{if } s \text{ is in a loop,} \\ \{[y, s', E \setminus \{(s, s')\}, t] \mid (s, s') \in E\}. \end{cases}$$

- **Remove Tail:** a complementary operation that moves the tail to its predecessor while removing the appropriate edge. We use $[y, s, E, t]\ominus$ to represent this operation.
- **Prepend Head:** connects the current head with a new head by adding the appropriate edge, and returns a single access graph:

$$f \oplus [y, s, E, t] := [y, f, E \cup \{(f, s)\}, t]$$

If the field graph was empty, the operation returns the access graph $[y, f, \varnothing, f]$.

- **Append Tail:** appends the field to the end of the access graph.

$$[y, s, E, t] \oplus f := [y, s, E \cup \{(t, f)\}, f].$$

- **Concatenate:** appends the field graph from the second operand to the first one. Both field graphs must not be empty and the base is taken from the left operand.

$$[a, s_1, E_1, t_1] \uplus [b, s_2, E_2, t_2] = [a, s_1, E_1 \cup E_2, t_2]$$

All of those operations additionally perform a cleanup to remove nodes which no longer reside on a path between the head and tail nodes.

3 Boomerang

A Boomerang query is denoted by $(s, \widetilde{a})$ – compute pointer information for the access graph $\widetilde{a}$ at the program statement s. Boomerang performs a staged computation: a *backward* pass followed by a *forward* pass. For both passes, Boomerang uses the IFDS framework to ensure flow- and context-sensitivity and exploit the performance gains of using point-wise procedure summaries. The backward pass traverses the control-flow graph (CFG) backwards to discover the possible allocation sites that the queried access graph $\widetilde{a}$ may point to. Starting at those statements, the forward pass then collects the access graphs that, at the query statement s, may also point to the same allocation sites (i.e., may alias with $\widetilde{a}$). In this section, we assume a query to be *context-free* – propagation is only allowed within the transitively reachable callees of the method where the query is issued. In Section 4.2, we describe how allocation sites within callers are detected by the combination of multiple context-free queries. By that, Boomerang only generates the part of the supergraph required to compute results and, if available, reuses parts of the supergraph it computed from previous queries. We use $\overset{b}{\rightarrow}$ to denote the backward path edges and $\overset{f}{\rightarrow}$ to denote the edges of the forward pass.

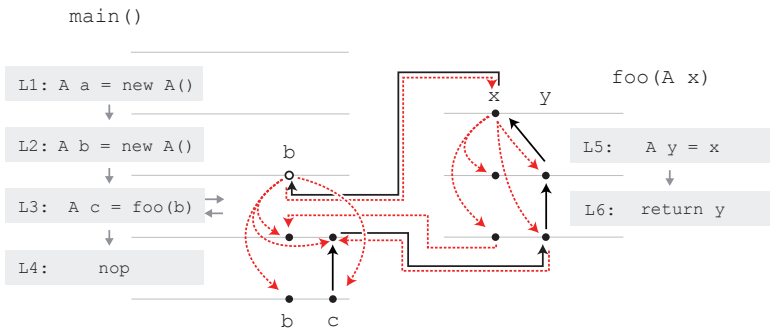


Figure 4 Constructing the exploded supergraph for the example in Figure 2.

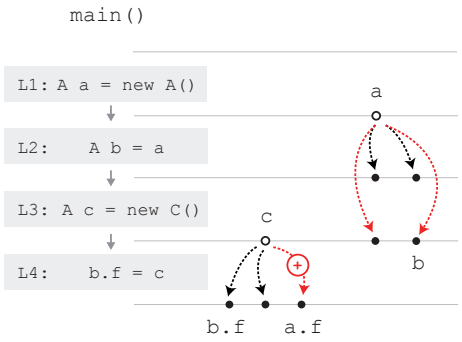


Figure 5 Handling field writes in BOOMERANG.

3.1 Basic Example

Figure 4 shows, for the example in Figure 2, the parts of the exploded super graph that have to be constructed to answer the query $(\mathbf{L4}, \mathtt{c})$. At first, BOOMERANG starts a backward analysis (solid edges) that follows the definition chain of `c`, which results in analyzing the method `foo`. When the backward analysis reaches the allocation site at **L2**, BOOMERANG starts a forward analysis (dotted edges) that backtracks the path previously taken by the backward analysis. BOOMERANG discovers `b` and `c`, and reports that those variables may alias as they may both point to the object allocated at **L2**.

In BOOMERANG, the forward path edges are designed to hold the points-to and alias information. In the example, the forward path edges at statement **L4** are $\langle \mathbf{L2}, \mathtt{b} \rangle \xrightarrow{f} \langle \mathbf{L4}, \mathtt{b} \rangle$ and $\langle \mathbf{L2}, \mathtt{b} \rangle \xrightarrow{f} \langle \mathbf{L4}, \mathtt{c} \rangle$. Both edges share the same origin, $\langle \mathbf{L2}, \mathtt{b} \rangle$ and by construction the data-flow facts of the targets may alias. Hence, at statement **L4** the variables `b` and `c` may alias and both are allocated at **L2**. To compute the alias information of `c`, BOOMERANG does not construct the part of the exploded super graph concerning variable `a`, avoiding the unnecessary effort that a whole-program analysis would make. As a side effect, this computation creates reusable backward and forward summaries for `foo`.

3.2 Handling Field Accesses

The flow functions of a precise pointer analysis are non-distributive [7]. They introduce a level of *indirection* that occurs, for instance, at *field write* statements. In Figure 5, to precisely reason about all aliases of `b.f` at the write at **L4**, the analysis must also take into

```
1: procedure BOOMERANG(s, v̂)
2:     P̄ = PROPAGATE(⟨s, v̂⟩ →^b ⟨s, v̂⟩)                    ▷ backward analysis
3:     do
4:         p = P̄.pop()
5:         N̄ = HANDLEPOI(p)                                ▷ triggers forward or backward analyses
6:         P̄.addAll(N̄)
7:     while P̄ ≠ ∅
8:     return Res(s, v̂)
9: end procedure
```

Figure 6 The main algorithm of BOOMERANG.

account the aliases of the the base variable `b`. In that case, alias information can only be retrieved indirectly. This is an example of a point of indirection, denoted by $\mathcal{POI}$. We discuss points of indirection in more detail in Section 4.1.

A sound treatment of those indirections raises the need for an *outer fixed-point iteration* around the forward and backward propagations. Such a handling of the $\mathcal{POI}s$ allows BOOMERANG to reuse summaries within the inner, distributive propagations, while at the same time computing a solution to a problem that is non-distributive as a whole. Figure 6 presents the pseudo-code for the outer fixed-point iteration. The process bootstraps the inner backward analysis with an initial self-loop (using the IFDS method PROPAGATE [17]). The propagated edge triggers the backward solver to find the object allocations that may flow into $\widetilde{v}$. The flow functions discover any $\mathcal{POI}s$ they find along the way. This yields the initial set $\overline{P}$. Once the backward propagation is finished, each of the newly discovered $\mathcal{POI}$ is handled by HANDLEPOI. Depending on the type of the $\mathcal{POI}$, the appropriate handler is executed. These handlers can trigger forward or backward analyses and can find new $\mathcal{POI}s$ (denoted $\overline{N}$) that are added to the set $\overline{P}$. This is repeated until a fixed point is reached.

In Figure 5, for the query of `c` at **L4**, let us assume that the backward analysis has already discovered the allocation site at line **L3**. In the forward pass, `c` flows to `b.f` at the field write statement in **L4** —a $\mathcal{POI}$. In HANDLEPOI, a sub-query to BOOMERANG then requests all aliases of `b` at **L3**, which returns the set $\{\texttt{a}, \texttt{b}\}$. Finally, the original query is continued by adding the dashed forward path edge marked with $\odot$ due to the discovered alias `a`.

When all $\mathcal{POI}s$ have been discovered and handled, the outer algorithm returns $Res(s, \widetilde{v})$, the solution to the query. In the example from Figure 5, the query $(\mathbf{L4}, \texttt{c})$ results in the mapping $\langle \mathbf{L3}, \texttt{c} \rangle \mapsto \{\texttt{c}, \texttt{a.f}, \texttt{b.f}\}$. Thus, `a.f`, `b.f` and `c` all alias with each other, as their allocation site is statement **L3**. Section 3.4 describes the construction of $Res(s, \widetilde{v})$ in more detail.

3.3 Flow Functions

In BOOMERANG, flow functions are of type $\mathcal{S} \times \mathcal{APG} \to 2^{\mathcal{APG}}$. They take an access graph and a statement as input, and output a set of access graphs. We represent a flow function for the backward problem as $[\![s]\!]_b(\widetilde{\alpha})$ where s is a statement and α is an access graph. A flow function for the forward problem is similarly represented by $[\![s]\!]_f(\widetilde{\alpha})$.

The analysis is conducted on a three-address-code representation in which each statement contains at most one field dereference. We only define the flow functions for statements that affect the access graph being propagated as shown in Table 7a. To simplify the notations, we assume that invoke statements have at most one parameter.

Statement	Notation
Allocation site	$x \leftarrow \textbf{new}$
Assign statement	$x \leftarrow y$
Field read	$x \leftarrow y.f$
Field write	$x.f \leftarrow y$,
Invoke statement	$r \leftarrow c.m(p)$

(a) Handled Statements.

$\mathcal{POI}$	Example	Symbol	Analysis
Allocation site	`a = new`	[**Alloc**]	Backward
Field read	`a = b.f`	[**Read**]	Backward
Alias on call	`c.m(args)`	[**Call**]	Backward
Alias on return	`c.m(args)`	[**Return**]	Forward
Field write	`a.f = b`	[**Write**]	Forward

(b) Points of indirection.

Figure 7 The statements handled by BOOMERANG and the points of indirection.

The definition of the flow functions is straightforward and simply follows the assignments of variables. Evaluating a flow function, causes new $\mathcal{POI}s$ to be discovered (shown next to the flow functions definitions). The $\mathcal{POI}s$ are handed over to the outer fixed-point iteration for later processing. Table 7b enumerates the types of $\mathcal{POI}s$ and which analysis discovers them. Section 4 describes how the $\mathcal{POI}s$ are processed in the outer fixed-point iteration.

3.3.1 Backward Analysis

For simplicity, we use $\widetilde{\alpha}$ as the argument of the flow function and express $\widetilde{\alpha}$ as $[v, s, E, t]$. We use $\widetilde{\alpha}_{>0}$ for the cases where the field graph cannot be empty, and $\widetilde{\alpha}_{=0}$ when the graph must be empty.

Normal-Flow Functions

$$[\![x \leftarrow y]\!]_b(\widetilde{\alpha}) = \begin{cases} \{[y, s, E, t]\} & \text{if } v = x \\ \{\widetilde{\alpha}\} \end{cases}$$

$$[\![x.f \leftarrow y]\!]_b(\widetilde{\alpha}) = \begin{cases} \ominus[y, s, E, t] & \text{if } v = x \wedge s = f \\ \{\widetilde{\alpha}\} \end{cases}$$

$$[\![x \leftarrow \textbf{new}]\!]_b(\widetilde{\alpha}) = \begin{cases} \varnothing & \text{if } v = x \wedge \widetilde{\alpha}_{=0} \quad [\textbf{Alloc}] \\ \varnothing & \text{if } v = x \wedge \widetilde{\alpha}_{>0} \\ \{\widetilde{\alpha}\} \end{cases}$$

$$[\![x \leftarrow y.f]\!]_b(\widetilde{\alpha}) = \begin{cases} f \oplus [y, s, E, t] & \text{if } v = x \quad [\textbf{Read}] \\ \{\widetilde{\alpha}\} \end{cases}$$

At assignment statements of the form $x \leftarrow y$ where the access graph's local variable is x, the flow function replaces the base value of the access graph with y and keeps the whole field graph. For a field write statement $x.f \leftarrow y$, if the local variable and the first field of the access graph matches $x.f$, the base of the access graph is replaced by y and its first field is removed. At a field read statement, in contrast, the field f is prepended to the field graph. Additionally, if the flow functions discover $\mathcal{POI}s$ of type [**Read**] or [**Alloc**], they are added to the outer algorithm of BOOMERANG for later processing.

Call-Flow Functions

$$[\![r \leftarrow c.m(p)]\!]_b(\widetilde{\alpha}) = \begin{cases} \{[\texttt{retVal}, s, E, t]\} & \text{if } v = r \quad [\textbf{Call}] \\ \{[\texttt{this}, s, E, t]\} & \text{if } v = c \wedge \widetilde{\alpha}_{>0} \quad [\textbf{Call}] \\ \{[\texttt{arg}, s, E, t]\} & \text{if } v = p \wedge \widetilde{\alpha}_{>0} \quad [\textbf{Call}] \\ \varnothing \end{cases}$$

In a backward analysis, the call-flow functions transfer facts from the call site to all possible return statements of the callees. In BOOMERANG, the backward call-flow functions map all access graphs $\widetilde{a}$ for which the field graph is not empty. The base is then either the `this` variable or the argument to the callees. Java uses a call-by-value semantics, which is why a callee cannot redefine its parameters (including `this`). For this reason BOOMERANG propagates parameters only with non-empty field graphs. As a side effect, the backward call-flow function further registers a [**Call**].

Return-Flow Functions

$$[\![r \leftarrow c.m(p)]\!]_b(\widetilde{\alpha}) = \begin{cases} \{[c, s, E, t]\} & \text{if } v = \texttt{this} \wedge \widetilde{\alpha}_{>0} \\ \{[p, s, E, t]\} & \text{if } v = \texttt{arg} \wedge \widetilde{\alpha}_{>0} \\ \varnothing \end{cases}$$

The return-flow functions map the access graphs back to the appropriate scope at the call sites. The access graphs representing the `this` value or a parameter that does not have any field accesses are not mapped back to the caller scope. They are handled by the call-to-return flow functions.

Call-to-Return Flow Functions

$$[\![r \leftarrow c.m(p)]\!]_b(\widetilde{\alpha}) = \begin{cases} \varnothing & \text{if } v = r \\ \varnothing & \text{if } v \in \{c, p\} \wedge \widetilde{\alpha}_{>0} \\ \{\widetilde{\alpha}\} \end{cases}$$

Every access graph that is propagated by a call-flow function to hold within the callee is killed in the corresponding call-to-return flow function. These access graphs naturally flow out of the callees at the call site (via the return-flow function), if they are neither allocated nor overwritten within callees.

3.3.2 Forward Analysis

The forward flow functions are used to find all access graphs pointing to a given allocation site. The forward analysis is bootstrapped at an allocation site with an access graph having the allocated variable as base and an empty field graph.

Normal-Flow Functions

$$[\![x \leftarrow y]\!]_f(\widetilde{\alpha}) = \begin{cases} \{[x,s,E,t], \widetilde{\alpha}\} & \text{if } v = y \\ \varnothing & \text{if } v = x \\ \{\widetilde{\alpha}\} & \end{cases}$$

$$[\![x.f \leftarrow y]\!]_f(\widetilde{\alpha}) = \begin{cases} \{\widetilde{\alpha}\} \cup f \oplus [x,s,E,t] & \text{if } v = y \quad [\textbf{Write}] \\ \varnothing & \text{if } v = x \wedge s = f \\ \{\widetilde{\alpha}\} & \end{cases}$$

$$[\![x \leftarrow \textbf{new}]\!]_f(\widetilde{\alpha}) = \begin{cases} \varnothing & \text{if } v = x \\ \{\widetilde{\alpha}\} & \end{cases}$$

$$[\![x \leftarrow y.f]\!]_f(\widetilde{\alpha}) = \begin{cases} \{\widetilde{\alpha}\} \cup \ominus[x,s,E,t] & \text{if } v = y \wedge s = f \\ \varnothing & \text{if } v = x \\ \{\widetilde{\alpha}\} & \end{cases}$$

Whenever an assignment is encountered and its right-hand side matches the current fact, a new access graph with the prefix of the left-hand side is derived and propagated. In the forward flow functions, the access graph $\widetilde{\alpha}$ flowing into the function is maintained, as long as the left-hand side of an assign statement does not override it. $\mathcal{POI}s$ of type [**Write**] are detected at each field write statement.

Call-Flow Functions

$$[\![r \leftarrow c.m(p)]\!]_f(\widetilde{\alpha}) = \begin{cases} \{[\texttt{this}, s, E, t]\} & \text{if } v = c \\ \{[\texttt{arg}, s, E, t]\} & \text{if } v = p \\ \varnothing & \end{cases}$$

In the forward analysis, the call-flow functions map facts from call sites to the start points of the target methods. It is important to note that the access graphs with an empty field graph and having variable c or any parameter p as their base are also propagated into the callees. These access graphs can generate aliases within those callees. This is different from the backward analysis where facts that do not involve fields are not propagated into the callees.

Return-Flow Functions

$$[\![r \leftarrow c.m(p)]\!]_f(\widetilde{\alpha}) = \begin{cases} \{[c, s, E, t]\} & \text{if } v = \texttt{this} \quad [\textbf{Return}] \\ \{[p, s, E, t]\} & \text{if } v = \texttt{arg} \quad [\textbf{Return}] \\ \{[r, s, E, t]\} & \text{if } v = \texttt{retVal} \quad [\textbf{Return}] \\ \varnothing & \end{cases}$$

The return-flow functions map the base of the access graph back to the appropriate variable in the calling-context. As we will see later in Section 4, a $\mathcal{POI}$ of type [**Return**] needs to be handled here because of possible new aliases in the new scope.

Call-to-Return Flow Functions

$$[\![r \leftarrow c.m(p)]\!]_f(\widetilde{\alpha}) = \begin{cases} \varnothing & \text{if } v = r \\ \varnothing & \text{if } v \in \{c, p\} \wedge \widetilde{\alpha}_{>0} \\ \{\widetilde{\alpha}\} & \end{cases}$$

In the call-to-return flow function, an access graph is killed if it has a non-empty field graph and its base is the call receiver or a parameter. Such access graphs are handled by the call-flow functions and flows into the callees.

3.4 Path Edges

In IFDS, a path edge is a general summary between a fact at the method's first statement and an arbitrary node of the exploded supergraph belonging to the same method. In Boomerang, we define path edges as IFDS path edges between any two arbitrary nodes of the same method. The first statement restriction is loosened to be able to encode points-to information in the path edges. For example, when the backward analysis finds an allocation site ([**Alloc**] in the backward normal-flow functions), then the path edge $\langle \texttt{a = new}, \texttt{a}\rangle \xrightarrow{f} \langle \texttt{a = new}, \texttt{a}\rangle$ is added to the forward analysis. During the forward propagation where IFDS extends the added path edge, the origin of the path edge $\langle \texttt{a = new}, \texttt{a}\rangle$ is maintained for the derived path edges in order to keep track of the new allocation site.

For each derived forward path edge $\langle s, \widetilde{\alpha}\rangle \xrightarrow{f} \langle t, \widetilde{\beta}\rangle$, Boomerang ensures that the object that is accessible through the access graph $\widetilde{\alpha}$ at statement s is also accessible through $\widetilde{\beta}$ at statement t. By design in IFDS, the path edge is restricted to the same method (i.e., the statements s and t belong to the same method). To track inter-procedural information, Boomerang defines three types of path edges depending on the access graph $\widetilde{\alpha}$ and the statement s:

- **Direct** ($s = d \leftarrow new$)
  The statement s is an allocation site of the access graph $\widetilde{\alpha}$. Therefore, $\widetilde{\alpha} = d$. If the forward solver generates such an edge, then the allocated object flows to the access graph $\widetilde{\beta}$ at statement t, where both s and t are in the same method.
- **Transitive** ($s = r \leftarrow c.m(p)$)
  The statement s is a call site. Therefore, the access graph referred to by $\widetilde{\alpha}$ is allocated in a method that is transitively reachable through this call. Upon exiting the callee, the forward path edge propagated in the caller has the call site s as its source statement. Boomerang can therefore distinguish between multiple objects instantiated at the same statement, but reachable via two distinct calls in the same method.
- **Parameter** ($s \in startPoint$)
  The statement s is the first statement of the method containing t. The base variable of the access graph $\widetilde{\alpha}$ represents a formal parameter of the method. This means that the allocation site of the access graph $\widetilde{\alpha}$ is not found in the current method nor any of its transitively reachable callees; the access graph has been allocated in a context that is not yet known to the analysis. This type of edge corresponds to the path edge in IFDS used to generate intra-procedural summaries [12].

Once the outer fixed-point algorithm of Boomerang stabilizes, the result $Res(t, \widetilde{\gamma})$ of the query $(t, \widetilde{\gamma})$ is gathered from all generated forward path edges. It is a map with domain $\{\langle s, \widetilde{\alpha}\rangle \mid \exists \langle s, \widetilde{\alpha}\rangle \xrightarrow{f} \langle t, \widetilde{\gamma}\rangle\} \subseteq \mathcal{S} \times \mathcal{APG}$. To each element $\langle s, \widetilde{\alpha}\rangle$ of the domain, Boomerang

associates the set $\{\widetilde{\beta} \mid \exists \langle s, \widetilde{\alpha}\rangle \xrightarrow{f} \langle t, \widetilde{\beta}\rangle\} \subseteq \mathcal{APG}$. The statements of the elements of the domain hold the allocation sites, whereas the accompanying sets contain all access graphs accessible at statement t which point to the related allocation site. Hence, any access graph $\widetilde{\alpha}$ in one of the sets may-aliases with $\widetilde{\gamma}$ at statement t. We note that by $\widetilde{\alpha} \blacktriangleleft Res(t, \widetilde{\gamma})$.

The domain of the result can contain origins of parameter path edges. This means that some allocation sites were undetected, as they are contained in the callers. Further queries can be issued to find them. Therefore, the results are independent of any calling-context. This design decision enables BOOMERANG to answer a query under particular client-provided calling-context. We address this feature in Section 4.2.

4 Implementation Challenges

In this section, we discuss how BOOMERANG addresses the following challenges:

- How are the $\mathcal{POI}s$ handled?
- How can the analysis answer queries under specific client-provided context?

4.1 Points of Indirection

As explained in Section 3.2, $\mathcal{POI}s$ are resolved in the outer fixed-point iteration. Each $\mathcal{POI}$ is treated independently.

4.1.1 Allocation Site [Alloc]

Upon discovering an allocation site during the backward pass, BOOMERANG needs to determine which access graphs might point to it. From an allocation site $a \leftarrow new$, BOOMERANG creates an access graph $\widetilde{\alpha}$ with base a and an empty field graph. Then, the path edge $\langle a \leftarrow new, \widetilde{\alpha}\rangle \xrightarrow{f} \langle a \leftarrow new, \widetilde{\alpha}\rangle$ is added to the work list of the forward solver. To avoid unnecessary computations (e.g., in a branch of non-interest to the query), the forward propagations are directed along the path(s) taken by the backward analysis which discovered the $\mathcal{POI}$. This is done, by following the backward path edges. If an additional backward path is discovered later on, the appropriate forward part will then be computed.

4.1.2 Field Write Statements [Write]

Let $\langle r, \widetilde{\alpha}\rangle \xrightarrow{f} \langle t, \widetilde{\beta}\rangle$ be the path edge currently processed by the forward solver when the [**Write**] is discovered. The statement t is a (forward) successor statement of a field write statement $(x.f \leftarrow y)$ where the base of the access graph $\widetilde{\beta}$ is x and its head is f. BOOMERANG searches for aliases of x, the base of the overwritten field. For all aliases $\widetilde{\delta} \blacktriangleleft Res(t, x)$, we construct $\widetilde{\gamma} = \widetilde{\delta} \uplus \widetilde{\beta}$. For each $\widetilde{\gamma}$, BOOMERANG adds the path edge $\langle r, \widetilde{\alpha}\rangle \xrightarrow{f} \langle t, \widetilde{\gamma}\rangle$ to the IFDS work list of the forward solver. For example, in Figure 5, the path edge $\langle \mathbf{L3}, \texttt{c}\rangle \xrightarrow{f} \langle \mathbf{L4}, \texttt{a.f}\rangle$ (marked with $\odot$) is added to the forward solver because before **L4**, the base variable `b` and `a` are aliases.

4.1.3 Alias on Forward Return-Flow [Return]

The [**Return**] indirection occurs upon a change of scope. This change occurs when a path edge reaches a method's exit statement and the analysis propagation is continued in its caller or when an existing summary is applied at a call site [17]. Figure 8 illustrates the former case.

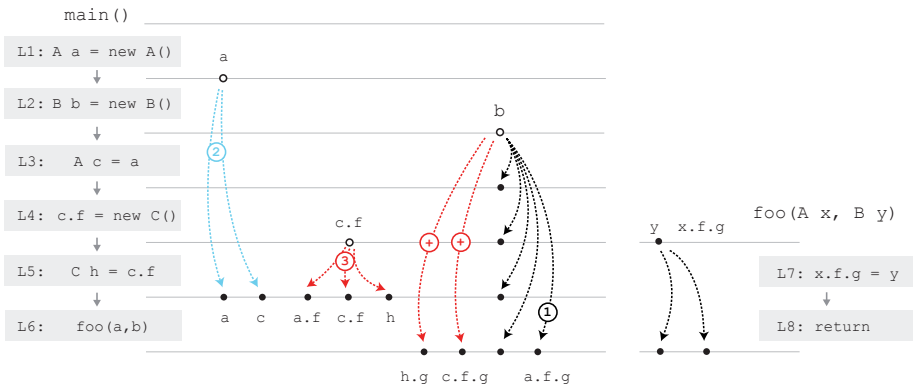


Figure 8 Handling alias on forward return-flow [**Return**].

Let us assume that we are interested in the query $(\mathbf{L6}, \mathtt{b})$. The forward propagation of the allocation site at **L2** constructs the edge $\langle \mathbf{L2}, \mathtt{b}\rangle \xrightarrow{f} \langle \mathbf{L6}, \mathtt{a.f.g}\rangle$ (labeled with ①). This edge in the `main` method is a result of the propagations in `foo`. At this point, BOOMERANG needs to be aware of new aliases in the new scope. This requires a recursive query for all prefixes of `a.f.g`. First, the query $(\mathbf{L5}, \mathtt{a})$ is issued, delivering the alias $c \blacktriangleleft Res(\mathbf{L5}, \mathtt{a})$ (leftmost path edges marked with ②). The field `f` is then appended to all access graphs of the result set. This triggers the query $(\mathbf{L5}, \mathtt{c.f})$, resulting in the path edges ③, that eventually finds the alias `h`. Finally, BOOMERANG appends the field `g` to the results and propagates the edges $\langle \mathbf{L2}, \mathtt{b}\rangle \xrightarrow{f} \langle \mathbf{L6}, \mathtt{c.f.g}\rangle$ and $\langle \mathbf{L2}, \mathtt{b}\rangle \xrightarrow{f} \langle \mathbf{L6}, \mathtt{h.g}\rangle$ forward. In Figure 8, those are the edges labeled with ⊕.

Formally, BOOMERANG iterates as follows. Let us assume the path edge $\langle u, \widetilde{\alpha}\rangle \xrightarrow{f} \langle t, \widetilde{\beta}\rangle$, where $\widetilde{\beta} = [a, k, E, l]$, and t, one successor of the call site s, is propagated in the caller. For each node j of the access graph $\widetilde{\beta}$, the set F_j is computed:

$$F_j := \begin{cases} \{\widetilde{\gamma} \oplus k \mid \widetilde{\gamma} \blacktriangleleft Res(s, a)\} & \text{if } j = k, \\ \{\widetilde{\gamma} \oplus j \mid \widetilde{\gamma} \blacktriangleleft Res(s, \widetilde{\delta}),\ \widetilde{\delta} \in F_i, (i, j) \in E\}. \end{cases}$$

By construction, the set F_j depends on the set F_i where i is a predecessor node of j in the access graph. The first set to be constructed is the set F_k, as it does not depend on any other sets. All other sets are computed successively[1]. For the tail node l of $\widetilde{\beta}$, the set F_l eventually represents all other aliases of $\widetilde{\beta}$ that hold just before the call site. For all of those aliases, the path edges $\langle u, \widetilde{\alpha}\rangle \xrightarrow{f} \langle t, \widetilde{\delta}\rangle$ with $\widetilde{\delta} \in F_l$ are propagated forward to connect the indirect aliases occurring in the new scope. In the example of Figure 8, the constructed sets are $F_{\mathtt{f}} = \{\mathtt{a.f}, \mathtt{c.f}\}$ and $F_{\mathtt{g}} = \{\mathtt{a.f.g}, \mathtt{c.f.g}, \mathtt{h.g}\}$.

4.1.4 Field Read Statements [Read]

Handling field read statements is the inverse of forward handling field write statements. At a [**Read**] point, we can assume a path edge $\langle r, \widetilde{\alpha}\rangle \xrightarrow{b} \langle t, \widetilde{\beta}\rangle$ where t is a (backward) successor of a field read statement $(x \leftarrow y.f)$. The access graph $\widetilde{\beta}$ then has y as its base variable, and the field f as its head. Due to the field read, the allocations to any alias of $y.f$ become allocation

[1] The computation might lead to a fixed-point iteration over the sets F_j.

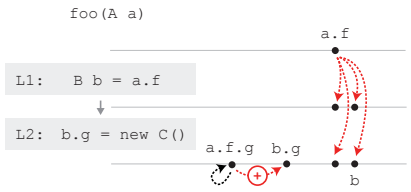


Figure 9 Example for [**Call**]

sites of interest for the backward analysis. To detect these allocation sites, BOOMERANG first searches aliases of y. The original access graph $\widetilde{\beta}$ is then concatenated to each alias $\widetilde{\delta} \blacktriangleleft Res(t, y)$ to form $\widetilde{\gamma} = \widetilde{\delta} \uplus \widetilde{\beta}$. The newly formed access graphs $\widetilde{\gamma}$ are propagated backward by adding the path edges $\langle r, \widetilde{\alpha} \rangle \xrightarrow{b} \langle t, \widetilde{\gamma} \rangle$ to the backward solver.

4.1.5 Alias on Backward Call-Flow [Call]

This $\mathcal{POI}$ is discovered in the call-flow functions of the backward analysis. Similar to [**Return**], [**Call**] leads to a recursive query of the prefixes of the access graph entering the method. Figure 9 shows an example where the backward analysis searches for the allocation site of `a.f.g` within `foo`. This access graph is allocated through an alias within `foo`. The local alias `b` of `a.f` has to be found first by executing the query (**L2**, `a.f`). This query results in the red dotted path edges shown on the right. From these path edges, BOOMERANG derives that it also needs to find allocations of `b.g` as well as `a.f.g`. Hence, the backward path edge labeled with $\odot$ is added to the backward solver. This edge leads to the detection of the allocation ([**Alloc**]) of the alias at **L2**.

In general, at a [**Call**], an access graph $\widetilde{\alpha} = [a, k, E, l]$ enters a method via the call-flow function. The last field of this access graph is then removed using the remove tail operation, resulting in the set $[a, k, E, l]\ominus$. For each element $\widetilde{\beta}$ of that set, all aliases are computed. The last step is to re-append the field l using the append tail operation ($\oplus$) and continue the backward propagation with the obtained aliases.

4.2 Client-Driven Context-Resolution

Up to this section, a query issued within method `m` was *context free*, causing it to propagate into `m` and its callees, but not into its callers. This design enables the creation of a client-driven context-resolution system where, after the context-free query is resolved, the client can define which particular calling contexts are relevant for BOOMERANG to further explore. This excludes contexts that are uninteresting to the client, helping increase precision and improve performance. If the client always chooses all available contexts, BOOMERANG computes pointer information over all contexts, like a standard pointer analysis would.

To use the client-driven context-resolution system, the client defines a *calling context*, which is a statement (optionally enriched by client-specific information) and a *calling-context function*, a function $\mathcal{C} \to 2^{\mathcal{C}}$, where $\mathcal{C}$ is the set of calling contexts. With the help of this function, BOOMERANG internally constructs a *context graph* $\mathcal{G}$, whose nodes are calling contexts. Initially, $\mathcal{G}$ is composed of one node, the *initial calling context*, given by the statement at which the query is issued. BOOMERANG then extends the graph successively. Except for the initial calling context, the other calling contexts' statements are call sites.

A context-free query is associated with each calling context of the context graph. Until a fixed point is reached, BOOMERANG computes the given queries for the calling contexts.

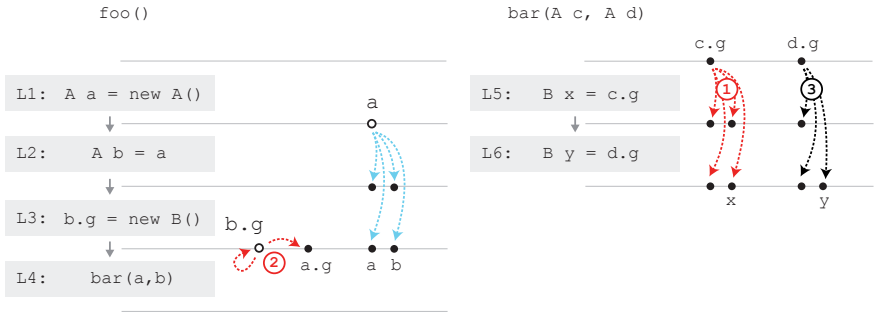


Figure 10 Client-Driven Context-Resolution.

This is done by using a worklist where each workslist item is a pair of a calling context and a query, $[C, (s, \widetilde{\alpha})]$.

Processing an item computes $Res(s, \widetilde{\alpha})$, the given context-free query. The result is then associated to the elements' calling context C. A check is performed to determine if the result set could potentially be missing allocation sites or aliases due to callers. This happens if the result contains an access graph $\widetilde{\beta}$ with a formal argument of the method as its base. If so, BOOMERANG evaluates the calling context function by supplying it the current calling context C to obtain more contexts. Each access graph $\widetilde{\beta}$ is then mapped to each new calling context D, and the element $[D, (t, \widetilde{\gamma})]$ is added to the worklist. Hereby, t is the (call site) statement of the calling context D and $\widetilde{\gamma}$ is the access graph $\widetilde{\beta}$ with replaced formals by actual arguments of the call site. In addition to extending the context graph from callee to callers, some results at the nodes must be *pushed back* along the caller to callee relationships within the graph, meaning that appropriate worklist elements are added. We clarify this in the following example. Finally, once the fixed point is reached, BOOMERANG extracts the complete pointer information from the context graph, by traversing the graph from each context-node containing an allocation site, to the original client context.

Figure 10 illustrates the context-resolution algorithm. The client analysis wishes to solve the query $(\mathbf{L6}, \mathtt{x})$ in method `bar` under *all calling contexts*. First, the context graph only contains the node **L6** [2] and the worklist is initialized with the element $[\mathbf{L6}, (\mathbf{L6}, \mathtt{x})]$. The result of the context-less query $(\mathbf{L6}, \mathtt{x})$ is $\langle \mathbf{SP}_{\mathtt{bar}}, \mathtt{c.g} \rangle \mapsto \{\mathtt{x}, \mathtt{c.g}\}$, which is encoded by the path edges ending at **L6** and labeled with ①. As the path edges are of parameter type, the alias information holds if and only if there exists an allocation site for `c.g` in one of `bar`'s callers. BOOMERANG then queries the client for the potential calling contexts of the calling context **L6**, and receives the set $\{\mathbf{L4}\}$. Therefore the calling context **L4** is created and linked to **L6** in the context graph, and $[\mathbf{L4}, (\mathbf{L4}, \mathtt{a.g})]$ is added to the worklist. For this query, BOOMERANG delivers $\langle \mathbf{L3}, \mathtt{b.g} \rangle \mapsto \{\mathtt{a.g}, \mathtt{b.g}\}$, —the path edges marked with label ② [3].

The results computed at the calling context **L4** in `foo` are pushed back to the callee `bar`, as the context graph contains an edge from **L6** to **L4**. This maps the access graphs `a.g` to `c.g` and `b.g` to `d.g`. At this point, BOOMERANG knows that `c.g`, `x`, and `d.g` are allocated at **L3**. However, one alias, `y`, is still missing. Therefore, an additional element is added to the worklist: $[\mathbf{L6}, (\mathbf{L6}, \mathtt{d.g})]$. It creates the path edges labeled with ③. Finally, the path

[2] We assume no client-specific information is bound to a context.
[3] The query internally creates the dotted unlabeled path edges within a sub-query to find the aliases `a` and `b`.

edges ①, ②, and ③ prove the reachability between the allocation site at **L3** and the access graph in $\{\texttt{c.g}, \texttt{x}, \texttt{d.g}, \texttt{y}\}$, which is the result of our initial query under all calling contexts.

In the example from Figure 10, a calling context is solely defined by a statement, and based on that the results are merged. Enriching the calling contexts with additional client-specific information enables the client to control the context-sensitivity of the results. For example, the client can add a call-stack of length k to make the analysis k-context-sensitive. As an alternative, the context can be modeled as a *value context*, holding more information about the (client's) data-flow value under which the method has been reached. We describe such value contexts in our experiments.

By filtering calling contexts, the client analysis can limit Boomerang's search space (e.g., by analyzing only `foo`, in Figure 10, and not all of the other callers of `bar`). This leads to significant savings in computational resources for Boomerang, as we show in **RQ3** in our experiments. The obtained results can, of course, only be considered sound with respect to the calling contexts provided by the client.

5 Evaluation

We implemented Boomerang on top of the Soot analysis framework and extended the existing IFDS solver Heros [4] to fit our needs (e.g., allowing path edges to start from a custom statement). The implementation was rigorously tested with more than 100 test cases. Boomerang is currently single-threaded, but a multi-threaded version is in the works.

5.1 Research Questions

Our empirical evaluation aims to answer the following research questions:

RQ1. How precise is Boomerang compared to other pointer analyses?
RQ2. How does the use of Boomerang affect the performance of a client analysis?
RQ3. How much influence does the client-driven context-resolution have on the search space for Boomerang?

5.2 Demand-driven Pointer Analyses

In **RQ1** and **RQ2**, we compare Boomerang to two flow-insensitive demand-driven pointer analyses: the refinement-based points-to analysis by Sridharan and Bodík [23] and the alias analysis by Yan et al. [30] (hereafter denoted SB and DA, respectively).

A query (m, v) to SB comprises a local variable v and the method m it belongs to. The query result is a points-to set of the given variable with each allocation site being enriched by a calling context. A query for DA consists of two local variables and the methods they belong to: (m_1, v_1, m_2, v_2). The result is a boolean value stating whether the two variables v_1 and v_2 in the given methods may alias or not. Although this interface is convenient for some client analyses (e.g., datarace analysis), points-to sets cannot be derived from alias information.

5.3 Results

RQ1. How precise is Boomerang compared to other pointer analyses?

Due to the difference in analysis types, returned information and query format, we compare the precision of SB, DA, and Boomerang on the common basis of alias information. We

use DA's query format as this information can be derived by all three analyses. For SB, an alias query is mapped to two points-to queries, one for each variable of the alias query. If the intersection of the two points-to sets is non-empty, the two variables may alias. Boomerang, with its rich results, only needs to issue one query for one of the two variables. If the second variable is in the result set, both may alias. To evaluate the precision of Boomerang's points-to information, we additionally compare its results to SB's points-to sets.

The PointerBench Micro-Benchmark. Evaluating the precision of a pointer analysis requires a ground truth to compare against. To the best of our knowledge, there exists no benchmarks for pointer analyses. In this paper, we introduce PointerBench[4], a micro-benchmark for pointer analyses that contains 36 specially crafted small programs that depict common pointer analysis issues for Java (e.g., handling collections, field-sensitivity, access paths). Each program is provided with may (and, whenever possible, must) information about all aliases, non-aliases, and allocation sites of a particular variable. Using PointerBench, alias, points-to, or any kind of pointer analysis can be compared against one another based on the same ground truth.

Alias Pairs. In Table 1, we report the true positives, false positives, and false negatives for each pair of aliasing/non-aliasing variables. Across all the programs in PointerBench, Boomerang achieves 100% recall and 90% precision, DA achieves 95% recall and 82% precision, and SB achieves 97% recall and 87% precision with respect to alias pairs.

The main reason for the loss of recall for DA and SB is the test case `AccessPath`. A limitation of the two analyses is that they only support queries on local variables. Therefore, it is not possible to check if two access graphs `a.f.g` and `b.h.i` alias. Additionally, DA reports a false negative for the test case `ObjectSensitivity`. This test case creates an object and passes it to an static method that returns its parameter (i.e., it is an identity function). In the caller of that function, the argument to the call and the return value should alias. DA incorrectly models static methods, which leads to this false negative.

In the group `Collections`, the test cases contain operations on `HashSet` and `HashMap` (e.g., inserting and retrieving elements). These collections store elements in arrays, but all three analyses are array-insensitive. Therefore, arrays are treated as a single assignment to the heap. All three analyses encounter a precision loss here.

For the test case `FlowSensitivty`, DA and SB report a false positive since they are both flow-insensitive. An additional false positive for DA is reported on the test case `Null`. Given the statements `a = null; b = a;`, DA reports `a` and `b` as an alias pair, even though the points-to sets of `a` and `b` are empty. We call this the *null-aliasing property*.

Allocation Sites. For each program in PointerBench, the last two columns of Table 1 show false negatives, false positives and true positives in terms of allocation sites reported by Boomerang and SB. The alias analysis DA has been excluded from this part of the evaluation as it does not produce any points-to information.

Across all test cases in PointerBench, Boomerang and SB achieve 100% and 98% recall, respectively. Similar to the aliasing case, SB cannot handle the `AccessPath` case. With respect to precision, Boomerang and SB achieve 86% and 69%, respectively. The main reason for SB's drop in precision are the test cases `Interprocedural` and `FieldSensitivity` due to SB being flow-insensitive.

[4] Available for download at `https://github.com/secure-software-engineering/PointerBench`

Table 1 The precision and recall of Boomerang, SB, and DA with respect to alias pairs and allocation sites on PointerBench. ⊖= false negatives, ⊕= false positives, ✓= true positives.

		Alias pairs			Allocation sites	
	Tests	Boomerang	DA	SB	Boomerang	SB
Basic	SimpleAlias	✓✓	✓✓	✓✓	✓	✓
	Loops	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓
	Interprocedural	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓	✓✓⊕⊕⊕⊕
	Parameter	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓	✓✓
	Recursion	✓	✓	✓	✓	✓
	ReturnValue	✓✓✓✓✓	✓✓✓✓✓⊕	✓✓✓✓✓	✓✓✓	✓✓✓⊕
	Branching	✓✓	✓✓	✓✓	✓✓	✓✓
General Java	AccessPath	✓✓	⊖⊖	⊖⊖	✓	⊖
	ContextSensitivity	6×✓	6×✓	6×✓	6×✓	6×✓
	FieldSensitivity	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓	✓✓⊕⊕⊕⊕
	FlowSensitivity	✓	✓⊕	✓⊕	✓	✓⊕
	ObjectSensitivity	✓✓✓✓	✓✓✓⊕⊕⊖	✓✓✓✓	✓✓	✓✓⊕
	StrongUpdate	✓✓✓⊕	✓✓✓⊕	✓✓✓⊕	✓✓⊕⊕	✓✓⊕⊕
Corner Cases	Exception	✓✓✓	✓✓✓⊕	✓✓✓⊕	✓✓	✓✓⊕
	Interface	✓✓	✓✓	✓✓	✓	✓
	Null		⊕			
	OuterClass	✓✓⊕	✓✓⊕	✓✓⊕	✓⊕	✓⊕
	StaticVariables	✓✓	✓✓	✓✓	✓	✓
	SuperClasses	✓✓⊕	✓✓⊕	✓✓⊕	✓⊕	✓⊕
Collections	Array	✓✓⊕	✓✓⊕	✓✓⊕	✓⊕	✓⊕
	List	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓	✓✓
	Map	✓✓⊕	✓✓⊕	✓✓⊕	✓⊕	✓⊕
	Set	✓⊕⊕	✓⊕⊕	✓⊕⊕	✓⊕	✓⊕
	Precision	0.90	0.82	0.87	0.86	0.69
	Recall	1.0	0.95	0.97	1.0	0.98

RQ2. How does the use of Boomerang affect the performance of a client analysis?

The usefulness of a support analysis is best evaluated in combination with a concrete client analysis. We evaluate the performance of FlowDroid, a context- and flow-sensitive taint analysis for Android [1]. We see FlowDroid as a representative of taint and other analyses of similar precision (e.g., typestate analysis). In this experiment, we compare the performance of FlowDroid on real-world applications from the Google Play Store when using Boomerang, SB and DA.

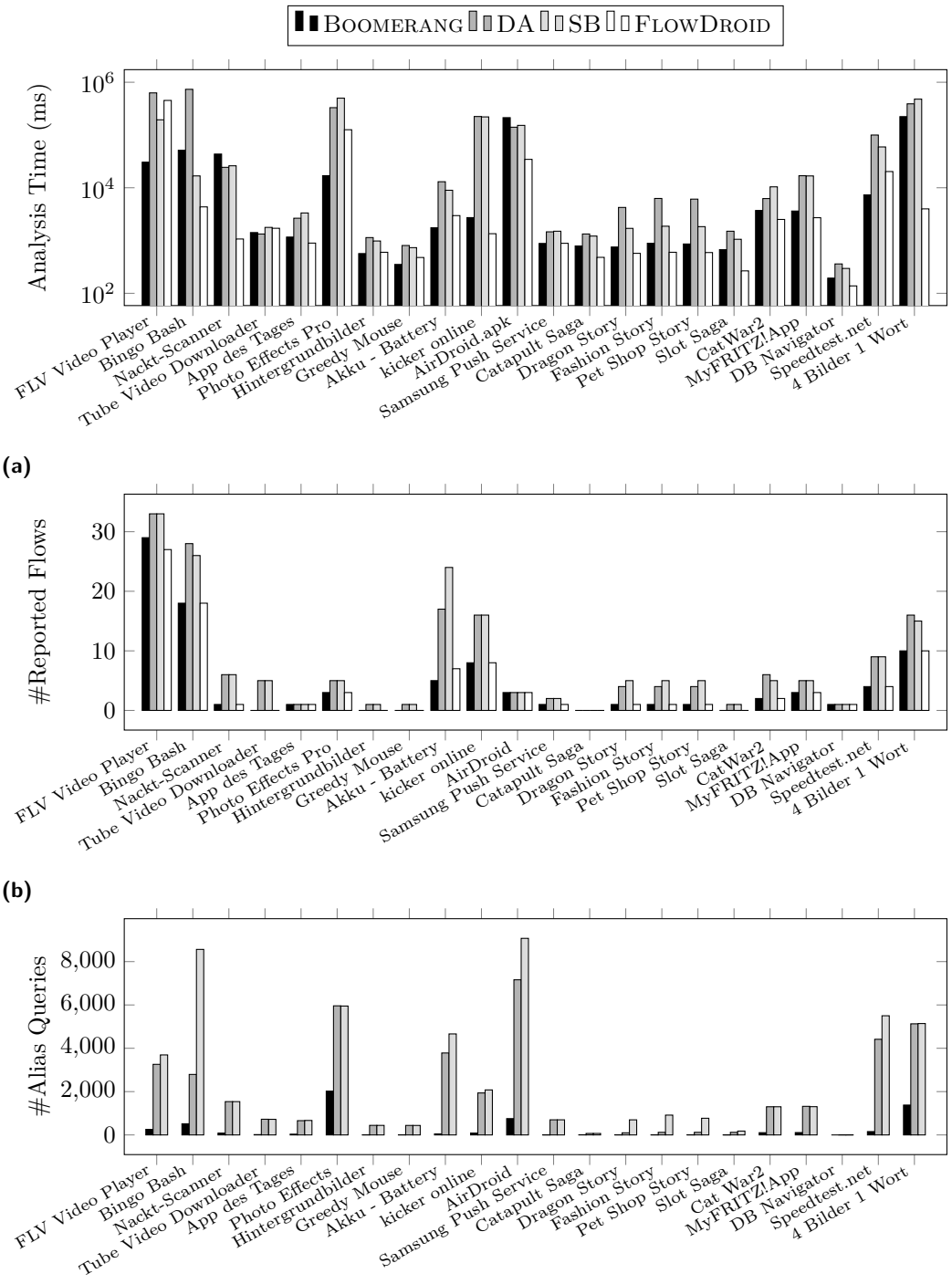
Like many taint or typestate analyses, FlowDroid requires alias information at field write statements and on return from callees to call sites. These are points of indirection where FlowDroid might indirectly taint other data-flow facts (FlowDroid access paths). At those statements, FlowDroid needs to obtain all aliases of the tainted access path. FlowDroid then taints those aliases and continues the taint propagation.

DA and SB's integration into FlowDroid need additional post-processing for the computation of all-alias sets, as there interface does not deliver the right information. To construct such a set A for a given FlowDroid access path `a.f.g`, all assign statements of the method containing the field-write statement or the call site are considered. For an assign statement, `c = d`, the analysis must check if the variable `d` aliases with `a`. In such case, the access path `c.f.g` is added to A. Similarly, for a field read statement `c = d.f`, where the field matches the first field of the access path, the analysis enquires whether `d` and `a` alias. If so, the access path `c.g` is also added to A. A field-write statement is handled likewise, but the appropriate field is prepended to the access path.

In contrast, when using Boomerang, FlowDroid requires only one query per field write or call site. Boomerang directly returns an alias set, whose access graphs only need to be translated to FlowDroid access paths. For the client-driven context-resolution system (Section 4.2) and the integration into FlowDroid, each calling context is enriched by one FlowDroid path edge. For the initial calling context, the edge that triggered the query is used. The calling-context function uses the *incoming map* [17] of FlowDroid's IFDS solver to compute the calling contexts. The incoming map describes how path edges in the caller are mapped to the path edges in the callee and guarantees the context-sensitivity for IFDS. In Boomerang, for a calling context enriched with a path edge p, for each relationship $q \to p$ of the incoming map, the calling-context function returns a calling context enriched by the path edge q at the appropriate call site. Boomerang then backtracks the path(s) FlowDroid's data-flow fact took and computes only the alias information necessary for the client. In the case of FlowDroid, calling contexts are *value contexts* since the calling context holds additional information about the data-flow value that reached the statement.

We set a timeout of one second per query for all three integrated pointer analyses. We also limit the overall analysis time for each application to 15 minutes. We ran FlowDroid with the three alias strategies, and, as a baseline, with FlowDroid's own intertwined alias strategy [1] on 100 randomly chosen apps from the top 500 apps of the Google PlayStore. The experiment was conducted using JDK version 1.7.0_85 on a 64 core (Intel Xeon E5-4640) with 32GB heap space.

FlowDroid supported by Boomerang successfully analyzed 53 applications within the overall time limit, while the support of DA, SB, and the original alias strategy of FlowDroid could only analyze 23, 30, and 43 applications, respectively. Figure 11a shows the analysis time of the 22 applications for which all four strategies did not time out. As the analysis times vary a lot, the diagram uses a logarithmic y-axis. On average, using Boomerang, FlowDroid is 2.6x faster than using DA and 3.9x faster than using SB. When comparing to FlowDroid, the original alias strategy performs 1.6x times better than



(a)

(b)

(c)

Figure 11 Comparison of the three alias strategies with respect to a) analysis time, b) reported flows, and c) triggered alias queries.

Boomerang. The reason is that FlowDroid's alias strategy also follows the incoming map [1].

Figure 11b shows that the choice of the alias strategy affects the number of taint flows reported by FlowDroid. In total, FlowDroid reports 92 flows with both Boomerang and the original FlowDroid aliasing strategy, whereas it reports 168 flows with DA and 174 flows with SB. When running the analyses on the FlowDroid test cases, we noticed that SB and DA reported many spurious flows. The main cause for those false positives is that both analyses are flow-insensitive and do not consider calling contexts. On average, the use of DA and SB cause FlowDroid to report 1.8x more flows when compared to Boomerang or the original FlowDroid aliasing strategy.

Figure 11c presents the total number of alias queries issued by FlowDroid for the 22 fully analyzed applications. The integration of FlowDroid with Boomerang issues an average of 19.9x fewer queries when compared to DA and 29.4x fewer queries when compared to SB. This shows that Boomerang issues significantly fewer queries, due to the richer pointer information Boomerang provides.

In summary, (1) using DA and SB, the FlowDroid analysis is slower, (2) more alias queries are triggered and (3) more taint flows are reported when compared to the integration with Boomerang.

RQ3. How much influence does the client-driven context-resolution have on the search space for Boomerang?

As described in **RQ2**, for the integration into FlowDroid, Boomerang uses the client-driven context-resolution system and follows the incoming map of FlowDroid's IFDS solver to find appropriate calling context at which Boomerang continues. Incorporating the context-resolution system, can be seen as a filtering of the call graph's edges before Boomerang continues with its computation in the callers. Hence, the filtering will reduce the search space for Boomerang in advance. For DA's and SB's integrations, all calling contexts are directly considered, even though they might be of non-interest to the client.

To evaluate the effectiveness of the filtering, we measured and report the following parameters for the 53 applications that Boomerang could analyze within the time limit:

- *Context requests*: the number of times Boomerang requests calling context information from FlowDroid,
- *All contexts*: the number of call sites in the call graph (hence, the number of calling contexts in which the analysis could continue) and
- *Client contexts*: the filtered number of contexts that FlowDroid returns to Boomerang.

In the worst case, FlowDroid provides all possible contexts. In each of the returned calling context, new queries to Boomerang are triggered (see Section 4.2). Boomerang then has to evaluate the queries for all of these calling contexts. This worst-case scenario is similar to the behavior of DA and SB; both propagate along all possible calling contexts.

Figure 12 shows the percentage of filtered calling contexts as a function of the number of context requests made for each application. The high ratio of filtering significantly reduces the search space for Boomerang in advance. We also observed that FlowDroid tends to filter out less calling contexts for applications where Boomerang requests fewer contexts. In summary, the consideration of the calling context of interest by the client early in the computation phase significantly reduces the search space of Boomerang.

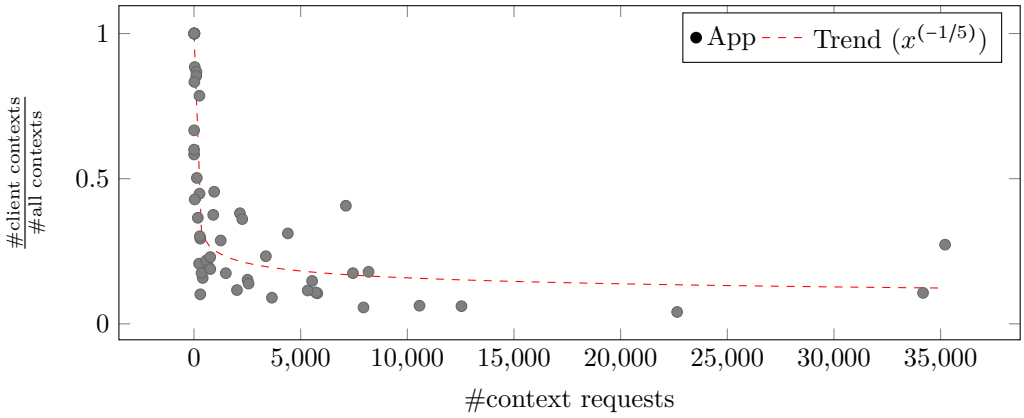


Figure 12 The relation between the number of context requests to the effectiveness of the pre-filter for Boomerang.

5.4 Discussion

Boomerang is particularly designed for demand-driven clients, such as FlowDroid, seeking high precision and requiring all-alias sets. Boomerang is well-suited for object-tracking analyses, such as taint or typestate analyses, as it makes those clients easier to implement, and returns results faster, and with a higher precision than existing approaches.

If client analyses require simpler points-to or alias information, they might not benefit from using Boomerang. For example, for a race-checking client analysis, the alias analysis DA seems to be a more reasonable choice than Boomerang because for such a client a boolean answer to the question whether two given variables alias is usually sufficient. In future work, we plan to test the applicability of Boomerang as a general-purpose pointer analysis for other types of client analyses and also invite others to do so. As others have argued before [11], the choice of the optimal pointer analysis heavily depends on the client. But it is exactly for this reason that we advocate to evaluate future pointer analyses also with respect to how clients would need to query them and how that impacts the performance of those clients.

6 Related Work

Pointer analyses provide information about which pointers point-to which memory locations in a given program. Such information is necessary for many static client analyses such as escape analysis [20], shape analysis [8], and call graph construction [9, 26, 27]. This demand causes significant effort towards developing various pointer-analysis techniques. Ryder [21] provides a comprehensive taxonomy of many dimensions that affect the precision and cost of a given pointer analysis. Sridharan et al. [24] present a survey of the state-of-the-art alias analyses for object-oriented programs. For the purpose of this work, we categorize previous work on pointer analysis into two broad themes: *whole-program pointer analyses* and *demand-driven pointer analyses*.

6.1 Whole-Program Pointer Analyses

Lhoták et al. [14] developed Spark, a flexible framework for points-to analysis of Java programs. Spark provides a Soot [29] transformation that constructs the call graph of the input program on-the-fly while calculating the points-to sets. Spark's analysis is context- and flow-insensitive. Paddle [2] is a drop-in replacement for Spark that allows for a context-sensitive but flow-insensitive whole-program points-to analysis using binary-decision diagrams.

Bravenboer et al. [5] present Doop, a Datalog-based points-to analysis framework for Java programs. Doop can also construct the call graph on-the-fly while computing the points-to sets. Doop offers various context-sensitivity configurations, such as call-site-sensitive, object-sensitive, and heap-abstraction-sensitive. Doop can support limited, method-local flow-sensitivity by performing the pointer analysis on static single assignment (SSA) form. Some support for flow-sensitivity is in preparation.

De et al. [6] propose a whole-program flow- and context-sensitive analysis that uses access paths. They use a similarly rich representation as Boomerang, allowing for strong updates. The approach is based on the call-strings approach and was evaluated with a call-string length of 1. This approach quickly becomes imprecise in contrast to the functional approach we use in Boomerang.

6.2 Demand-Driven Pointer Analyses

Heintze et al. [10] introduce a demand-driven approach for pointer analysis that performs just enough computation to determine the points-to sets for a given list of pointer variables. However, the information obtained is context- and flow-insensitive.

Sridharan et al. [25] show that points-to information can be encoded as a context-free language (CFL). The problem of on-demand pointer analysis can then be re-defined as a CFL-reachability problem that can be solved on a graph representation. Labeling the edges of the graph enables field-sensitivity. As opposed to Boomerang, this formulation is flow-insensitive.

Based on this CFL-reachability, Shang et al. [22] present DynSum, a demand-driven context-sensitive but flow-insensitive points-to analysis. DynSum uses method summaries to improve performance for multiple queries within the same method. The answer to a query to DynSum is a points-to set. As we have previously discussed, a result to Boomerang encodes alias information, as well as points-to sets.

Yan et al. [30] propose a novel on-demand alias analysis for Java that is formulated as a CFL-reachability problem. For methods with a high in-degree (a preset threshold) in the call graph, summaries for the alias information are computed. Because it is an alias analysis, no points-to sets are generated and also not all aliasing variables can be directly retrieved as it is possible with Boomerang.

7 Conclusion

We have presented Boomerang, a demand-driven pointer analysis that provides rich pointer information to the client, comprising all allocation sites the queried pointer can point to, as well as access graphs representing all pointers that, at the queried statement, may point-to that same allocation site.

The analysis uses two instantiations (backward/forward) of the IFDS framework. Its procedure summaries provide efficiency, while flow-sensitivity and unlimited context-sensitivity

provide high precision. An additional outer fixed-point iteration within Boomerang yields field-sensitivity and soundiness [15]. The recursive evaluation of the backward and forward-analysis passes is carefully coordinated to construct only the minimal part of the exploded super graph necessary to answer a given query. Boomerang further allows a client to specify calling contexts to tailor the computational effort to specific parts of interest.

We have shown that Boomerang is more precise than the state-of-the-art demand-driven analyses on PointerBench. We have also shown that using Boomerang, clients compute their results issuing fewer points-to analysis queries, which significantly accelerates the overall analysis.

Acknowledgements. We would like to thank all anonymous reviewers and our shepherd Atanas Rountev to improve this work. This work was supported by the Fraunhofer Attract program and the DFG Collaborative Research Center CROSSING.

References

1 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, page 29, 2014.

2 Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *PLDI*, pages 103–114. ACM Press, 2003.

3 Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *ICSE*, pages 5–14, 2010.

4 Eric Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *SOAP 2012*, pages 3–8, July 2012.

5 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, pages 243–262, 2009.

6 Arnab De and Deepak D'Souza. Scalable flow-sensitive pointer analysis for Java with strong updates. In *ECOOP*, pages 665–687, 2012.

7 Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *PLDI*, pages 230–241, 1994.

8 Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL*, pages 1–15, 1996.

9 David Grove and Craig Chambers. A framework for call graph construction algorithms. *TOPLAS*, 23(6), 2001.

10 Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *PLDI*, pages 24–34, 2001.

11 Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, pages 54–61, 2001.

12 Susan Horwitz, Thomas W. Reps, and Shmuel Sagiv. Demand interprocedural dataflow analysis. In *FSE*, pages 104–115, 1995.

13 Amey Karkare, Amitabha Sanyal, and Uday P. Khedker. Heap reference analysis for functional programs. *CoRR*, 2007.

14 Ondrej Lhoták and Laurie J. Hendren. Scaling Java points-to analysis using SPARK. In *CC*, pages 153–169, 2003.

15 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, January 2015.

16 Nomair A. Naeem and Ondrej Lhoták. Typestate-like analysis of multiple interacting objects. In *OOPSLA*, pages 347–366, 2008.

17 Nomair A. Naeem, Ondrej Lhoták, and Jonathan Rodriguez. Practical extensions to the IFDS algorithm. In *CC*, pages 124–144, 2010.

18 Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *POPL*, pages 327–338, 2007.

19 Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

20 Cristina Ruggieri and Thomas P. Murtagh. Lifetime analysis of dynamically allocated objects. In *POPL*, pages 285–293, 1988.

21 Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *CC*, pages 126–137, 2003.

22 Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In *CGO*, pages 264–274, 2012.

23 Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.

24 Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 196–232. Springer, 2013.

25 Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *OOPSLA*, pages 59–76, 2005.

26 Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *OOPSLA*, pages 264–280, 2000.

27 Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA*, pages 281–293, 2000.

28 Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *FASE*, pages 210–225, 2013.

29 Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *CC*, pages 18–34, 2000.

30 Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for Java. In *ISSTA*, pages 155–165, 2011.

Transactional Tasks: Parallelism in Software Transactions

Janwillem Swalens[1], Joeri De Koster[2], and Wolfgang De Meuter[3]

1 **Software Languages Lab, Vrije Universiteit Brussel**
`jswalens@vub.ac.be`
2 **Software Languages Lab, Vrije Universiteit Brussel**
`jdekoste@vub.ac.be`
3 **Software Languages Lab, Vrije Universiteit Brussel**
`wdmeuter@vub.ac.be`

Abstract

Many programming languages, such as Clojure, Scala, and Haskell, support different concurrency models. In practice these models are often combined, however the semantics of the combinations are not always well-defined. In this paper, we study the combination of futures and Software Transactional Memory. Currently, futures created within a transaction cannot access the transactional state safely, violating the serializability of the transactions and leading to undesired behavior.

We define *transactional tasks*: a construct that allows futures to be created in transactions. Transactional tasks allow the parallelism in a transaction to be exploited, while providing safe access to the state of their encapsulating transaction. We show that transactional tasks have several useful properties: they are coordinated, they maintain serializability, and they do not introduce non-determinism. As such, transactional tasks combine futures and Software Transactional Memory, allowing the potential parallelism of a program to be fully exploited, while preserving the properties of the separate models where possible.

1998 ACM Subject Classification D.1.3 [Concurrent Programming] Parallel Programming; D.3.2 [Language Classifications] Concurrent, distributed, and parallel languages

Keywords and phrases Concurrency, Parallelism, Futures, Threads, Fork/Join, Software Transactional Memory

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.23

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.2.1.13`

1 Introduction

Concurrent programming has become essential to exploit modern hardware, especially since multicore processors have become ubiquitous. At the same time, programming with concurrency is notoriously tricky. Over the years, a plethora of concurrent programming models have been designed with the aim of combining performance with safety guarantees. In this paper, we study futures and Software Transactional Memory.

Futures are used to introduce parallelism in a program: the construct `fork` e starts the concurrent execution of the expression e in a *parallel task*, and returns a future [3, 16]. A *future* is a placeholder that is replaced with the value of e once its task has finished. Futures make it easy to add parallelism to a program [16] and can be implemented efficiently [8, 7, 19].

30th European Conference on Object-Oriented Programming (ECOOP 2016).
Editors: Shriram Krishnamurthi and Benjamin S. Lerner; Article No. 23; pp. 23:1–23:28



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

While futures enable parallelism, Software Transactional Memory (STM) allows access to shared memory between parallel tasks [26]. It introduces *transactions*: blocks in which shared memory locations can be read and modified safely. STM simplifies concurrent access to shared memory, as transactions are executed atomically and are guaranteed to be serializable.

It is often desirable to combine concurrency models and this is frequently done so in practice [27]. Languages such as Clojure, Scala, and Haskell provide support both for parallel tasks and transactions. However, the combination of these two models is not always well defined or supported. Using transactions to share memory between parallel tasks has a clearly defined semantics. Conversely, parallelizing a single transaction by creating new tasks is either not allowed (e.g. Haskell) or leads to unexpected behavior (e.g. Clojure and Scala). For example, in Section 3 a path-finding algorithm is shown in which several search operations are executed in parallel on a grid. In this application, each task starts a transaction that searches for a single path. We would like to further exploit the available parallelism in each individual search operation by starting new subtasks from within each path-finding transaction. This is one application where we show the need for the combination of both *transaction-in-future* and *future-in-transaction* parallelism.

In this paper, we present *transactional tasks*, a novel construct to introduce parallelism within a transaction (Section 4). A transaction can create several transactional tasks, which run concurrently and can safely access and modify their encapsulating transaction's state. We demonstrate several desirable properties of transactional tasks (Section 5). Firstly, transactional tasks are coordinated: the tasks in a transaction either all succeed or all fail. Secondly, the serializability of the transactions in the program is maintained. Thirdly, transactional tasks do not introduce non-determinism in a transaction.

Furthermore, we have implemented transactional tasks on top of Clojure (Section 6), and applied them to several applications from the widely used STAMP benchmark suite [21]. This confirms that they can produce a speed-up with limited developer effort (Section 7). As a result, we believe that transactional tasks successfully combine futures and Software Transactional Memory, allowing the potential parallelism of a program to be fully exploited, while preserving the properties of the separate models where possible.

2 Background: Parallel Tasks, Futures, and Transactions

We start by defining parallel tasks, futures, and transactions separately. We also provide a formalization for each model, list their properties and describe their use cases.[1]

The semantics described here are deliberately very similar to those offered both by Clojure and Haskell. We aim to approximate them closely, to demonstrate how the described problems also apply to these programming languages. A list of differences between our formal model and the implementations of Clojure and Haskell is given in Appendix A.

[1] We use the following notation for sets and sequences. $\uplus$ denotes disjoint union, i.e. if $A = B \uplus C$ then $B = A \backslash C$. We use a short-hand for unions on singletons: $A \cup a$ signifies $A \cup \{a\}$. The notation $S = a \cdot S'$ deconstructs a sequence S into its first element a and the rest S'. The empty sequence is written $[]$. We write $\overline{a}$ for a (possibly empty) sequence of a.

2.1 Parallel Tasks and Futures

2.1.1 Description and Properties

A **parallel task** or thread is a fragment of the program that can be executed in parallel with the rest of the program. A runtime system schedules these tasks over the available processing units. In a typical implementation, a distinction is made between the lightweight tasks created in the program, and more heavyweight OS threads over which they are scheduled. Managing and scheduling the threads is a responsibility of the runtime, and transparent to the program.

A parallel task can be created using `fork` e.[2] This begins the evaluation of the expression e in a new task, and returns a future. A **future** is a placeholder variable that represents the result of a concurrent computation [3, 16]. Initially, the future is *unresolved*. Once the evaluation of e yields a value v, the future is *resolved to* v. The resolution of future f can be synchronized using `join` f: if the future is still unresolved, this call will block until it is resolved and then return its value.

In a functional language, `fork` and `join` are **semantically transparent** [13]: a program in which `fork` e is replaced by e and `join` f is replaced by f is equivalent to the original. As a result, futures can be added to a program to exploit the available parallelism without changing its semantics. Programmers can wrap an expression in `fork` whenever they believe the parallel execution of the expression outweighs the cost of the creation of a task.

A common use case of futures is the parallelization of homogenous operations, such as searching and sorting [16]. For this purpose, Clojure provides a parallel map operation: `(pmap f xs)` will apply `f` to each element of `xs` in parallel. Futures are also used to increase the responsiveness of an application by executing long-running operations concurrently, e.g. in graphical applications expensive computations or HTTP requests often return a future so as not to block the user interface.

2.1.2 Formalization

Figure 1 defines a language with support for parallel tasks and futures. The syntax consists of a standard calculus, supporting conditionals (`if`), local variables (`let`), and blocks (`do`). To support futures, the syntax is augmented with references to futures (f), and the expressions `fork` e and `join` e.

The program state p consists of a set of tasks. Each task contains the expression it is currently evaluating, and the future to which its result will be written. The future f associated with each task is unique, and hence can be considered an identifier for the task. A program e starts with initial state $\{\langle f, e\rangle\}$, i.e. it contains one "root" task that executes e. We use evaluation contexts to define the evaluation order within expressions. The program evaluation context $\mathcal{P}$ can choose an arbitrary task, and use the term evaluation context $\mathcal{E}$ to find the active site in the term. $\mathcal{E}$ is an expression with a "hole $\square$". We note $\mathcal{E}[e]$ for the expression obtained by replacing the hole $\square$ with e in $\mathcal{E}$.

We define the operational semantics using transitions $p \rightarrow_{\mathrm{f}} p'$; the subscript f denotes all reduction rules that apply to futures. The operational semantics is based on [13] and [32]. Rule congruence$|_{\mathrm{f}}$ defines that the base language can be used in each task. Transitions in the base language are written $e \rightarrow_{\mathrm{b}} e'$ and define a standard λ calculus, but are not detailed here. The expression `fork` e creates a task in which e will be evaluated, and reduces to the

[2] In Clojure this is `(future e)`, in Scala `Future { e }`, in Haskell `forkIO e`.

Syntax

$c \in \text{Constant} \quad ::= \texttt{true} \mid \texttt{false} \mid 0 \mid 1 \mid \ldots$

$x \in \text{Variable}$

$f \in \text{Future}$

$v \in \text{Value} \quad ::= c \mid x \mid \lambda x.e \mid f$

$e \in \text{Expression} \quad ::= v \mid e\ e \mid \texttt{if}\ e\ e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{do}\ \overline{e};\ e \mid \texttt{fork}\ e \mid \texttt{join}\ e$

State

Program state $p \quad ::= \mathrm{T}$

Task $t \in \mathrm{T} \quad ::= \langle f, e \rangle$

Evaluation Contexts

$\mathcal{P} ::= \mathrm{T} \cup \langle f, \mathcal{E} \rangle$

$\mathcal{E} ::= \square \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \texttt{if}\ \mathcal{E}\ e\ e \mid \texttt{join}\ \mathcal{E} \mid \texttt{let}\ x = \mathcal{E}\ \texttt{in}\ e \mid \texttt{do}\ \overline{v};\ \mathcal{E}\ ;\overline{e}$

Reduction Rules

$$\text{congruence}|_{\mathrm{f}} \quad \frac{e \to_{\mathrm{b}} e'}{\mathrm{T} \cup \langle f, \mathcal{E}[e] \rangle \to_{\mathrm{f}} \mathrm{T} \cup \langle f, \mathcal{E}[e'] \rangle}$$

$$\text{fork}|_{\mathrm{f}} \quad \frac{f' \text{ fresh}}{\mathrm{T} \cup \langle f, \mathcal{E}[\texttt{fork}\ e] \rangle \to_{\mathrm{f}} \mathrm{T} \cup \langle f, \mathcal{E}[f'] \rangle \cup \langle f', e \rangle}$$

$$\text{join}|_{\mathrm{f}} \quad \mathrm{T} \uplus \left\langle f, \mathcal{E}[\texttt{join}\ f'] \right\rangle \uplus \left\langle f', v \right\rangle \to_{\mathrm{f}} \mathrm{T} \cup \langle f, \mathcal{E}[v] \rangle \cup \left\langle f', v \right\rangle$$

Figure 1 Operational semantics of a language with futures.

future f'. After the expression e has been reduced to a value v, `join` f' will also reduce to v. It is possible to join a task multiple times, each join reduces to the same value v. A join can only be reduced by rule join$|_{\mathrm{f}}$ if the corresponding future is resolved to a value, this detail encodes the blocking nature of our futures.

The semantic transparency property of futures is maintained by our semantics: when a task is created, its expression is evaluated and a placeholder f' is returned. When the task is joined, the placeholder is used to look up the value of the expression. This is equivalent to evaluating the future's expression in place: as our base language is purely functional, an expression always evaluates to the same value, regardless of the task it is executed in.

2.2 Software Transactional Memory

2.2.1 Description and Properties

Software Transactional Memory (STM) is a concurrency model that allows multiple threads to access shared variables, grouping the accesses into *transactions* [26]. In our case, these *transactional variables* are memory locations that contain a value, created using `ref` v. In Haskell these are called TVars, in Clojure they are refs. A transaction `atomic` e encapsulates an expression that can contain a number of *primitive operations* to the shared objects, such as reads (`deref` r) and writes (`ref-set` r v).

Transactional systems guarantee **serializability**: transactions appear to execute serially, i.e. the steps of one transaction never appear to be interleaved with the steps of another [18]. The result of a transactional program, which may execute transactions concurrently, must always be equal to the result of *a* serial execution of the program, i.e. one in which no transactions execute concurrently.

Transactions are used to allow safe access to shared memory in programs with multiple parallel tasks. These applications typically contain complex data structures of which pieces are encapsulated in transactional variables. For example, a web browser could encapsulate its Document Object Model in transactional variables, as its plug-ins are interested only in a subtree of the Document Object Model, but expect a consistent view of this part. Other examples include rich text documents, HTML pages, and CAD models [14]; networks, mazes,

Syntax

$r \in$ Reference

$v \in$ Value $::= \cdots \mid r$

$e \in$ Expression $::= \cdots \mid$ `ref` $e \mid$ `deref` e
$\mid$ `ref-set` $e\ e \mid$ `atomic` e

State

Program state $p ::= \langle \mathrm{T}, \theta \rangle$

Task $t \in \mathrm{T} ::= \langle f, e \rangle$

Heap $\theta \in$ Reference $\rightharpoonup$ Value

Local store $\tau \in$ Reference $\rightharpoonup$ Value

Evaluation Contexts

$\mathcal{P} ::= \langle \mathrm{T} \uplus \langle f, \mathcal{E} \rangle, \theta \rangle$

$\mathcal{E} ::= \cdots \mid$ `ref` $\mathcal{E} \mid$ `deref` $\mathcal{E}$
$\mid$ `ref-set` $\mathcal{E}\ e \mid$ `ref-set` $r\ \mathcal{E}$

Reduction Rules

$$\text{congruence}|_\mathrm{t} \quad \frac{\mathrm{T} \rightarrow_\mathrm{f} \mathrm{T}'}{\langle \mathrm{T}, \theta \rangle \rightarrow_\mathrm{t} \langle \mathrm{T}', \theta \rangle}$$

$$\text{atomic}|_\mathrm{t} \quad \frac{\langle \theta, \{\}, e \rangle \Rightarrow_\mathrm{t}^* \langle \theta, \tau', v \rangle}{\langle \mathrm{T} \uplus \langle f, \mathcal{E}[\texttt{atomic}\ e] \rangle, \theta \rangle \rightarrow_\mathrm{t} \langle \mathrm{T} \cup \langle f, \mathcal{E}[v] \rangle, \theta :: \tau' \rangle}$$

Transactional Transitions

$$\text{ref}||_\mathrm{t} \quad \frac{r \text{ fresh}}{\langle \theta, \tau, \mathcal{E}[\texttt{ref}\ v] \rangle \Rightarrow_\mathrm{t} \langle \theta, \tau[r \mapsto v], \mathcal{E}[r] \rangle}$$

$$\text{ref-set}||_\mathrm{t} \quad \langle \theta, \tau, \mathcal{E}[\texttt{ref-set}\ r\ v] \rangle \Rightarrow_\mathrm{t} \langle \theta, \tau[r \mapsto v], \mathcal{E}[v] \rangle$$

$$\text{deref}||_\mathrm{t} \quad \langle \theta, \tau, \mathcal{E}[\texttt{deref}\ r] \rangle \Rightarrow_\mathrm{t} \langle \theta, \tau, \mathcal{E}[(\theta :: \tau)(r)] \rangle$$

$$\text{atomic}||_\mathrm{t} \quad \langle \theta, \tau, \mathcal{E}[\texttt{atomic}\ e] \rangle \Rightarrow_\mathrm{t} \langle \theta, \tau, \mathcal{E}[e] \rangle$$

$$\text{congruence}||_\mathrm{t} \quad \frac{e \rightarrow_\mathrm{b} e'}{\langle \theta, \tau, \mathcal{E}[e] \rangle \Rightarrow_\mathrm{t} \langle \theta, \tau, \mathcal{E}[e'] \rangle}$$

Figure 2 Operational semantics for language with transactions.

graphs, and lists [21]. STM is especially suited if it is not yet known at the start of the atomic block which objects will be accessed: due to optimistic synchronization these are only locked for the duration of the commit.

2.2.2 Formalization

In Figure 2 we extend our language with support for transactions. We extend the syntax of Figure 1 with references to transactional variables (r), and the transactional operations. The program state is extended with a transactional heap θ that contains the values of the transactional variables.

Similar to the semantics for STM in Haskell [17], we make a distinction between two types of transitions. Regular program state transitions are written with a single arrow $p \rightarrow_\mathrm{t} p'$ and marked with $|_\mathrm{t}$. In contrast, transitions inside a transaction are written with a double arrow $\langle \theta, \tau, e \rangle \Rightarrow_\mathrm{t} \langle \theta, \tau', e' \rangle$ and marked with $||_\mathrm{t}$. Here, θ is the transactional heap at the start of the transaction, and remains unchanged throughout the transactional transitions as it is only used to look up values. Conversely, τ is the local store that contains the updates made to transactional variables during the transaction.

Transactional operations `ref` v creates a new r and sets its value in the local store to v. `ref-set` $r\ v$ updates the value of r to v. `deref` r will look for the value of r in $\theta :: \tau$, i.e. first in the local store τ and then the global heap θ.[3]

atomic$||_\mathrm{t}$ This rule applies to nested transactions, i.e. one `atomic` nested in another. A nested transaction is reduced in the context of its outer transaction, hence a nested `atomic` e reduces to e.

congruence$||_\mathrm{t}$ For now, we only allow reduction rules of the base language ($\rightarrow_\mathrm{b}$) to be applied in a transactional context. There is no congruence rule for $\rightarrow_\mathrm{f}$ in a transaction, so it cannot contain `fork` or `join` (this is introduced in the rest of this paper).

[3] $(\theta :: \tau)(r) = \begin{cases} \tau(r) & \text{if } r \in \mathrm{dom}(\tau) \\ \theta(r) & \text{otherwise} \end{cases}$

atomic$|_t$ The changes made to transactional variables during a transaction are gathered in τ, and applied to the heap at once, in atomic$|_t$. This encodes exactly the atomicity of a transaction: its effects appear to take place in one indivisible step. This also encodes the serializability of transactions: semantically it is as if each transaction were executed serially.

According to the classification of Moore and Grossman [22], our transactional semantics is high-level but small-step. It is *high-level*, as it relies on non-determinism in the reduction rules (specifically in the definition of $\mathcal{P}$) to find a successful execution. In contrast to an actual implementation, in the semantics there is no notion of conflicting or aborting transactions. Thus, we do not need to write down a commit protocol, allowing us to focus on the transitions inside the transaction. This simplification provides a straightforward model for programmers, and a correctness criterion against which more complex implementations can be verified. At the same time, the semantics is *small-step*: a transaction takes several steps to complete (denoted with $\Rightarrow_t$). This will allow us to describe parallelism in the transaction later.

In an actual implementation, multiple transactions execute in parallel. Clojure and Haskell implement STM using optimistic synchronization instead of (pessimistic) locks: a transaction is started, and if a conflict is detected while the transaction is running or during its commit, the changes are rolled back and the transaction is retried. Our implementation follows a similar approach and is described in detail in Section 6.

Transactions introduce non-determinism: a program with two tasks that each execute a transaction has two serializations, one in which the first transaction precedes the second and vice versa. However, this is a limited amount of non-determinism: the order in which the individual instructions of two transactions are interleaved does not matter, the developer only has to reason about transactions as a whole.

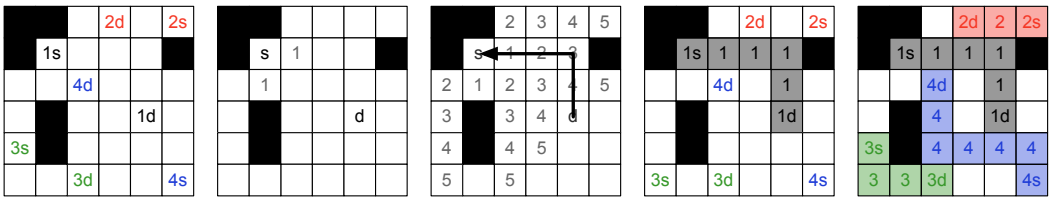
Lastly, by introducing transactions the `fork` and `join` constructs have lost their semantic transparency. For instance, a program that forks two tasks that each execute a transaction has a serialization in which the second transaction precedes the first. However, in the same program with the `fork` and `join` constructs removed the first transaction always precedes the second. This violates the semantic transparency property.

3 Motivation and Example

This section discusses the use of parallelism inside transactions, i.e. `fork` inside `atomic`. Using an example, we illustrate that it is desirable for certain applications to use parallelism in transactions. We discuss how this is realized in contemporary programming languages, and demonstrate that they do not provide satisfying semantics, as several desirable properties are not guaranteed.

As an example, we look at the Labyrinth application of the STAMP benchmark suite [21]. This application implements a transactional version of Lee's algorithm [20, 31]. Lee's algorithm, as it is used in chip design, places electric components on a grid and connects them without introducing overlapping wires.

Listing 1 shows the transactional version of Lee's algorithm implemented in Clojure, based on [31]. Its main data structure is `grid`, a two-dimensional array of transactional variables. The aim of the application is to find non-overlapping paths between given pairs of source and destination points. For example, Figure 3a depicts four source–destination pairs on an 6×6 grid, and Figure 3e shows a possible solution of connecting paths. The transactional variable `work-queue` is initialized to the input list of source–destination pairs.



(a) Global grid: input of four pairs of source and destination points

(b) Local grid of path 1: first expansion step

(c) Local grid of path 1: trace back from destination to source

(d) Global grid updated with path 1

(e) Final global grid with four non-overlapping paths

Figure 3 Different steps of the Labyrinth algorithm, illustrated using 4 paths on a two-dimensional 6×6 grid. The black squares are impenetrable 'walls'.

Listing 1 Transactional version of Lee's algorithm in Clojure.

```
1  (def grid (initialize-grid w h))      ; w × h array of cells, each cell is a ref
2  (def work-queue (ref (parse-input))) ; list of source–destination pairs
3
4  (loop [] ; [4]
5    (let [work (pop work-queue)] ; atomically take element from work queue
6      (if (nil? work)
7        true ; done
8        (do
9          (atomic
10           (let [local-grid (copy grid)
11                 [src dst]  work ; destructure source–destination pair using pattern matching
12                 reachable? (expand src dst local-grid)] ; ref-sets on local-grid
13             (if reachable?
14               (let [path (traceback local-grid dst)]
15                 (add-path grid path)))))                ; ref-sets on grid
16           (recur)))))
```

As long as the work queue is not empty, a source and destination pair in the input will be processed in a new transaction (lines 9–15). This happens in four steps. First, a local copy `local-grid` is created of the shared `grid` (line 10). Next, a breadth-first search expands from the source point (line 12), recording the distance back to the source in each visited cell of the `local-grid` using `ref-set` (Figure 3b), until the destination point is reached. Afterwards, the traceback function finds a path from the destination back to the source, minimizing the number of bends (line 14; Figure 3c). Finally, the shared `grid` is updated to indicate these cells are now occupied (line 15; Figure 3d). After the transaction has finished, this process is repeated until all work has been processed.[4]

To parallelize this algorithm, several "worker threads" execute the loop simultaneously. Each thread repeatedly takes a source–destination pair from the work queue and attempts to find a connecting path in a transaction. If two threads result in overlapping paths, a conflict occurs when updating the global `grid` (line 15), as the two threads attempt to write to the same transactional variable (that represents the cell where the paths collide). As a result, one of the two transactions is rolled back and will look for an alternative path.

[4] Clojure's loop `(loop [x 0] (recur 1))` defines and calls an anonymous function, in which `recur` executes a recursive call. It is equivalent to Scheme's named let: `(let n ([x 0]) (n 1))`.

Table 1 Characterization of the STAMP applications, abridged from [21]. These numbers were gathered on a simulated 16-core system. The transaction length and transactional execution time are color-coded high ●, medium ◐, low ○.

Application	Instructions /tx (mean)	Time in tx
labyrinth	219,571 ●	100% ●
bayes	60,584 ●	83% ●
yada	9,795 ●	100% ●
vacation-high	3,223 ◐	86% ●
genome	1,717 ◐	97% ●
intruder	330 ○	33% ◐
kmeans-high	117 ○	7% ○
ssca2	50 ○	17% ○

Minh et al. [21] measure various metrics of the applications in the STAMP benchmark, shown in Table 1. We compare the Labyrinth application with the other applications. Firstly we see that this application spends 100% of its execution time in transactions. Hence, the amount of parallelism in this program is maximally the number of transactions that are created, which is the number of input source–destination pairs, even on a machine with more cores. To allow more fine-grained parallelism to be exploited in this program, it is necessary to allow parallelism inside the transactions. Secondly, we infer that the transactions in this application take a long time to execute: an average transaction of the Labyrinth application contains several orders of magnitude more instructions than the other applications in the STAMP benchmark. This means conflicts will be costly: retrying a transaction incurs a large penalty. Parallelizing the transactions will reduce this cost.

Profiling reveals that, for typical inputs, more than 90% of the execution time of the program is spent in the expansion step, which performs a breadth-first search. Listings 2 and 4 show a simplified version of the relevant code. The `expand` function starts with a queue containing the `src` point (listing 2, line 2). In `expand-point` (listing 2, line 10), the first element of the queue is expanded, which updates the neighboring cells in `local-grid` for which a cheaper path has been found, using `ref-set` (listing 4, line 11), and returns these neighbors. These are then appended to the queue (listing 2, lines 8–10), and the loop is restarted. This continues until either the queue is empty or the destination has been reached.

In Listing 3, additional parallelism is exploited by replacing the breadth-first search algorithm by a parallel version of this algorithm. It uses *layer synchronization*, a technique in which all nodes of one layer of the breadth-first search graph are expanded—in parallel—before the next layer is started [33]. The queue now starts as a set containing only the `src` point (line 2). In each iteration of the loop, `expand-step` will expand all elements in the queue in parallel (lines 11–13), using Clojure's parallel map operation `pmap`. The union of all returned neighbors is then used as the queue for the next iteration of the loop (line 10). As before, this continues until either the queue is empty or the destination has been reached.

However, this code does not work as expected in Clojure! Each iteration of `pmap` is executed in a new thread, executing a call to `expand-point`, in which an `atomic` block appears. As transactions are thread-local in Clojure, it detects no transaction is running in the current thread, and starts a *new* transaction. When the `atomic` block ends, this inner transaction is committed. However, the surrounding transaction may still roll back, while the inner transaction cannot be rolled back anymore.

Listing 2 Expansion step: sequential, breadth-first search of local grid.

```
(defn expand [src dst local-grid]
  (loop [queue (list src)]
    (if (empty? queue)
      false ; no path found
      (if (= (first queue) dst)
        true ; destination reached
        (recur
          (concat
            (rest queue)
            (expand-point
              (first queue)
              local-grid)))))))
```

Listing 3 Parallel breadth-first search of local grid.

```
(defn expand [src dst local-grid]
  (loop [queue (set [src])}]
    (if (empty? queue)
      false ; no path found
      (if (contains? queue dst)
        true ; destination reached
        (recur (expand-step queue local-grid))))))

(defn expand-step [queue local-grid]
  (reduce union (set [])
    (pmap ; parallel map
      (fn [p] (expand-point p local-grid))
      queue)))
```

In general, Clojure allows threads to be created in a transaction, but they are not part of that transaction's context. When an `atomic` block appears in a new thread, a separate transaction is created with its own, possibly inconsistent, snapshot of the shared memory. This transaction will commit separately. Clojure does not consider thread creation as part of the transaction, hence it is not undone when the encapsulating transaction is rolled back. As such, *the serializability of the transactions is broken*. This is not the desired behavior of the presented example.

The same problems occur in most library-based STM implementations, including ScalaSTM, Deuce STM for Java, and GCC's support for transactional memory for C and C++.[5] Haskell, on the other hand, does not allow the code above to be written. The type system prohibits the creation of new threads in a transaction, as transactions are encapsulated in the STM monad while `forkIO` can only appear in the IO monad. As such, the serializability of transactions is guaranteed, but the parallelism is limited.

In conclusion, there are several reasons why current approaches to parallelism inside transactions are unsatisfactory [22]:

- By disallowing the creation of threads in transactions, in effect, **the parallelism of a program is limited**: every time transactions are introduced to isolate some computation from other threads, the potential performance benefits of parallelism *inside* this computa-

[5] `https://nbronson.github.io/scala-stm/`, `https://sites.google.com/site/deucestm/`, `https://gcc.gnu.org/wiki/TransactionalMemory`. GCC has support for transactional memory since version 4.7, although still labeled experimental.

Listing 4 Expand a point. (Code simplified for clarity.)

```
(defn expand-point [current local-grid]
  (atomic
    (let [cheaper-neighbors
            (filter
              (fn [neighbor]
                (< (cost neighbor current) ; cost of path to neighbor, through current
                   (deref neighbor)))      ; cost of previous path to neighbor
              (neighbors local-grid current))] ; neighbors of current
      (doseq [neighbor cheaper-neighbors]           ; for each cheaper neighbor:
        (ref-set neighbor (cost neighbor current))) ; set new cost
      cheaper-neighbors)))
```

tion are forfeited. This problem becomes apparent for programs containing long-running transactions. In the Labyrinth example, the maximal amount of parallelism is equal to the number of input source–destination pairs, even though additional parallelism could be exploited by the breadth-first search algorithm.

- In languages and libraries that do allow the creation of threads in transactions, **threads in transactions do not execute within the context of the encapsulating transaction**. This means they do not have access to the transactional state of the encapsulating transaction; instead a new transaction is started. The modifications of the new transaction are committed separately, and the serializability of transactions is no longer guaranteed.
- Inside a transaction, calling a library or other part of the program that contains `fork` is unsafe: it is either not allowed or can lead to incorrect results. This can severely **hinder re-usability** [15]. For instance, it is impossible to use a library that implements a parallel breadth-first search for the Labyrinth application.
- Last, transactions are used to ensure isolation, e.g. to prevent overlapping paths in the example, but they also form the unit of parallelism, evidenced by the fact that the maximal amount of parallelism is equal to the number of transactions. Moore and Grossman [22] and Haines et al. [15] argue that isolation and parallelism are orthogonal issues, but the **notions of isolation and parallelism are conflated**. Parallelizing the search algorithm should be orthogonal to the isolation between transactions, but it is not.

The ideal solution is one where several tasks can be created in a transaction and execute in parallel, i.e. allowing `fork` inside `atomic` (unlike Haskell's `forkIO`). Furthermore, these tasks should be able to access and modify the transactional variables, using the transactional context of the encapsulating transaction (unlike Clojure's or Scala's threads in transactions). With our approach, we want to allow coordination of different tasks by encapsulating them in a transaction: they either all succeed and all their changes are committed, or they all roll back. Finally, the serializability between all transactions in the program should be preserved.

4 Transactional Tasks

In this section, we define transactional tasks (Section 4.1). *Transactional tasks* are parallel tasks that are spawned in a transaction. A transactional task can access and modify the state of its encapsulating transaction, i.e. it operates within its *transactional context*. To allow multiple tasks to operate on the same transactional context simultaneously, each transactional task works on its own version of the data. This resembles traditional transactions, where each transaction operates on its own local store, and later commits its local updates. When a transactional task is joined by its parent, its updates are merged into its parent's context.

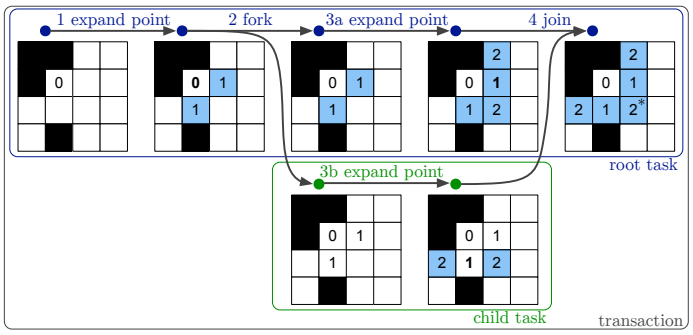


Figure 4 The timeline of the transactional tasks that are forked and joined when expanding the labyrinth grid. At each point in time, we show the grid as it exists in that task. The cells that are stored in the snapshot of the task are white, the modifications stored in the local store are blue.

We describe the joining semantics in Section 4.2, and discuss the properties of transactional tasks in Section 4.3.

4.1 A Transactional Task

In this section, we define what a transactional task is: what data it contains and what operations can be performed on it.

Each transaction starts with one *root* task that will evaluate the transaction's body. In a task, more tasks can be created using the `fork` *e* construct.

In contrast with other approaches, in our model each transactional task created by a fork also operates with respect to a transactional context. Conceptually, each transactional task operates on its own private copy of the transactional heap, and will access and modify that. To this end, a transactional task contains two parts: a *snapshot* containing the values of the transactional variables when the task was created, and a *local store* containing the values the task has written to transactional variables. In Figure 4, a timeline of the `expand` operation of the Labyrinth application is shown. At the start of the transaction, the snapshot of the root task contains the source cell at distance 0 (in white). After step 1, the local store is modified with the expanded cells at distance 1 (in blue).

When a task is created, its snapshot reflects the current state of the transactional variables. Hence, it is the snapshot of its parent task modified with the current local store of the parent. The snapshot of the root task is a copy of the transactional heap. The local store of a newly created task is empty. In Figure 4, step 2 creates a fork, the forked task's snapshot (in white) consists of its parent's snapshot combined with the parent's local store.

While a task executes, it can look up transactional values in its snapshot, and modify them by storing their updated values in the local store. When a task finishes its execution, its future is resolved to its final value. In steps 3a and 3b, the root task and its child both expand a cell and update their local stores (in blue).

When a task is joined for the first time, its local store is merged into the task performing the join, and the value of its future is returned. Step 4 copies the modified cells of the child task (blue cells) into the root task. Subsequent joins of the same task will not repeat this, as their changes are already merged; they will only return the last value of its future.

At the end of the transaction, the modifications of all transactional tasks should have been merged into the root task, and these are committed atomically. If a conflict occurs at commit time, the whole transaction is retried. If a conflict occurs in one of the tasks while

the transaction is still running, *all* tasks are aborted and the whole transaction is retried. In other words, the tasks within a transaction are coordinated so that they either all succeed or all fail: they form one atomic block.

4.2 Conflicts and Conflict Resolution Functions

On a join, conflicts are possible: it may happen that the child task has changed a transactional variable that the parent also modified since the creation of the child. In that case, a write–write conflict occurs. An example of such a conflict is marked on Figure 4 with an asterisk (*). Note that in this example both values happened to be the same, but that this is not necessarily the case in general.

For these situations, we allow a *conflict resolution function* to be specified per transactional variable (similar to Concurrent Revisions [9]). To this end, the programmer provides a `resolve` function when the transactional variable is created, i.e. `(ref initial-value resolve)`. If a conflict occurs, the new value of the variable in question is the result of $\texttt{resolve}(v_{\text{original}}, v_{\text{parent}}, v_{\text{child}})$, where v_{parent} and v_{child} refer to its value in the parent and child respectively, and v_{original} refers to its value when the child was created (stored in the child's snapshot).

In the Labyrinth example, the new value of a conflicting cell should be the minimum of the joining tasks, i.e. $\texttt{resolve}(o, p, c) = \min(p, c)$, as we want to find the cheapest path. Generally, conflict resolution functions are useful when each task performs a part of a calculation. For example, when each task calculates a partial result of a sum, the resolve function is $\texttt{resolve}(o, p, c) = p + c - o$, i.e. the total is the value in the parent plus the value that was added in the child since its creation. Similarly, if several tasks generate sets they are combined using $\texttt{resolve}(o, p, c) = p \cup c$, or if several tasks generate lists of results they can be combined with $\texttt{resolve}(o, p, c) = \text{concat}(o, p - o, c - o)$. If no conflict resolution function is specified, we default to picking the value in the child over the one in the parent, i.e. $\texttt{resolve}(o, p, c) = c$. We reason that explicitly joining a task is equivalent to requesting all its changes to be merged. On the other hand, the parent may be preferred by specifying $\texttt{resolve}(o, p, c) = p$.

Read–write "conflicts" are not considered to be actual conflicts in our model. If the parent reads a transactional variable while its child wrote to it, the parent still reads the 'old' value from its snapshot. The value will only be updated after an explicit `join` of the child. This prevents non-determinism, as the moment at which changes from one task become visible in another does not depend on how tasks are scheduled.

4.3 Properties

Transactional tasks provide several useful properties. We describe them here and discuss them more formally in Section 5.2.

Serializability of transactions Transactional tasks should always be joined before the end of the transaction they are created in. While a transaction can contain many tasks, each task is fully contained in a single transaction. The changes of the tasks in a transaction have therefore all been applied to the transaction's local store before commit, and on commit they are applied to the transactional heap at once. Consequently, transactions remain atomic and serializable. The introduction of transactional tasks does not change the developer's existing mental model of transactions.

Coordination of transactional tasks The changes of all tasks created in a transaction are committed at once. This means they either all succeed, or all fail. If a conflict occurs

in one task during its execution, all other tasks in the transaction are aborted and the transaction, as a whole, restarts. As such, transactions can be used as a mechanism to coordinate tasks. The developer can reason about transactions as one atomic block.

Determinism in transactions Transactional tasks do not introduce non-determinism in transactions. Firstly, it does not matter in which order the instructions of two tasks are interleaved since they both work on their own copies of the data. Secondly, the join operation is deterministic as long as the conflict resolution function is. The changes made in one task only become visible in another after an explicit and deterministic `join` statement, making the behavior in a transaction straightforward to predict. Furthermore, the developer can easily trace back the value of a variable by looking where tasks were joined.

Transactional tasks are nested transactions A transactional task makes a read-only snapshot of the transactional state upon its creation, and stores its modifications in a local store. This mirrors closely how a regular transaction creates a read-only copy of the transactional heap at its creation and stores its modifications in a local store. We could say that, while it is not syntactically shown, a transactional task starts a 'nested transaction'. This similarity should provide a familiar semantics to developers. The differences with nested transactions in the existing literature are discussed in Section 8.

No semantic transparency Transactional tasks are *not* semantically transparent: wrapping `fork` around an expression in a transaction changes the semantics of the program. As explained at the end of Section 2.2, this was already the case for non-transactional tasks that contain a transaction: they also do not necessarily evaluate to the same result every time.

Violating the semantic transparency is a necessary compromise to achieve our goal of executing tasks in parallel. If a transactional task were semantically transparent, its effects on the transactional state would need to be known at the point where it is created, before the parent task can continue. Therefore the child task and its parent would need to be executed sequentially. Instead, we opt to omit semantic transparency as a necessary compromise to accomplish the parallel execution of tasks.

Nonetheless, we argue that we maintain the "easy parallelism" of futures. Firstly, the determinism in transactions ensures that the order in which the instructions in the transactional tasks are interleaved does not affect the result. Secondly, transactional tasks provide a straightforward and consistent semantics of how the transactional effects of tasks are composed. Each task can modify the transactional state, but its effects become visible in a single step only when it is joined, and conflicts are resolved deterministically.

Non-transactional tasks maintain their semantics Tasks that are spawned outside a transaction are unchanged in our model: these *non-transactional tasks* cannot access or modify the transactional state directly. They can still create a transaction internally to modify the transactional state indirectly, as shown earlier in Section 2.

4.4 Summary

Using the concepts introduced in this section, the code in Listings 3 and 4 now behaves as expected. Each newly created task is part of the encapsulating transaction's context, and has access to its state. Transactional tasks can observe the changes that occurred before they were created, and they make their modifications in a private local store. When they are joined, their changes become visible in their parent task.

Eventually, all tasks are joined into the transaction and the transaction commits. All tasks of a transaction are coordinated, and transactions remain serializable, therefore all

tasks in one transaction behave as a single atomic block. Transactional tasks provide a straightforward semantics: behavior in a transaction is deterministic, and values can be traced back by looking at the `join` statements. Transactional tasks also provide a familiar semantics: they behave as nested transactions.

5 Semantics and Properties

In this section, we describe the semantics and properties of the transactional tasks model.[6]

5.1 Semantics

Figure 5 defines how transactional tasks are modeled. The program state and non-transactional tasks are as defined in Section 2. On the other hand, a transactional task t_x contains the following components: a future f used to join the task and read its final result, a snapshot σ that contains the values of the transactional variables at its start, a local store τ that stores its changes to transactional variables, a set F_s of spawned child tasks, a set F_j of tasks joined into this task, and finally the expression e being evaluated in this task.

Outside a transaction, rules are written $\langle T, \theta\rangle \to_{tf} \langle T', \theta'\rangle$, similar to Section 2.2. Inside a transaction, rules are written $T_x \Rightarrow_{tf} T'_x$, i.e. they work on a set of transactional tasks. A task belongs either to the set T if it is not transactional, or to one of the sets T_x if it is transactional (there is one such set for each transaction). Outside transactions, tasks keep their semantics as previously (rule congruence$|_{tf}$, as discussed in Section 2.1).

atomic$|_{tf}$ When a transaction is started, one transactional task is created: the "root" task. Its snapshot σ is the current state of transactional memory, i.e. the transactional heap θ. Its local store τ is initially empty. By applying one or more $\Rightarrow_{tf}$ rules, the transaction will eventually be reduced to another set of transactional tasks, one of which is the root task. Its local store has been updated to τ', which is merged into the transactional heap on commit. We require that all tasks created in the root task have been joined ($F_s \subseteq F_j$). This is discussed in further detail in Section 5.2.

The essence of this rule is the same as for atomic$|_t$ of Section 2.2: the changes τ' of the transaction are applied at once to the heap θ, which has not changed during the transaction. The major difference with rule atomic$|_t$ is that transactions now consist of a set of tasks, instead of one expression and local store.

congruence$||_{tf}$ Rule congruence$||_{tf}$ specifies that the transactional transitions from Figure 2 in Section 2.2 apply in transactional tasks as well. In other words, in a transactional task creating, reading, and writing transactional variables, nesting transactions, and using the base language work just like before—although now they operate on the task's local store.

fork$||_{tf}$ When a task is created, it creates a copy of the current state. Firstly, it creates a copy of the current transactional heap, i.e. its snapshot σ is set to the snapshot of its parent updated with the local modifications of its parent. In the actual implementation this is done differently for efficiency (see Section 6). Secondly, the set of joined tasks F_j is copied from its parent. This ensures that if any of these tasks are joined again, their transactional state is not merged again.

join$_1||_{tf}$ and join$_2||_{tf}$ Two rules describe the join operation of f' into f: join$_1||_{tf}$ is triggered on the first join, join$_2||_{tf}$ on subsequent joins.

[6] An executable implementation of the operational semantics built using PLT Redex [12] is available in the artifact, or at `https://github.com/jswalens/transactional-futures-redex`.

State

$$\begin{array}{rrl} \text{Program state} & p & ::= \langle \mathrm{T}, \theta \rangle \\ \text{Non-transactional task} & t \in \mathrm{T} & ::= \langle f, e \rangle \\ \text{Transactional task} & t_\mathrm{x} \in \mathrm{T_x} & ::= \langle f, \sigma, \tau, \mathrm{F_s}, \mathrm{F_j}, e \rangle \\ \text{Snapshot, local store} & \sigma, \tau \in \text{Reference} \rightharpoonup \text{Value} & \\ \text{Spawned and joined tasks} & \mathrm{F_s}, \mathrm{F_j} \subseteq \text{Future} & \end{array}$$

Reduction Rules

$$\text{congruence}|_{\mathrm{tf}} \quad \frac{\mathrm{T} \rightarrow_\mathrm{f} \mathrm{T}'}{\langle \mathrm{T}, \theta \rangle \rightarrow_{\mathrm{tf}} \langle \mathrm{T}', \theta \rangle}$$

$$\text{atomic}|_{\mathrm{tf}} \quad \frac{f' \text{ fresh} \qquad \{\langle f', \theta, \{\}, \varnothing, \varnothing, e \rangle\} \Rightarrow^*_{\mathrm{tf}} \{\langle f', \theta, \tau', \mathrm{F_s}, \mathrm{F_j}, v \rangle\} \cup \mathrm{T_x} \qquad \mathrm{F_s} \subseteq \mathrm{F_j}}{\langle \mathrm{T} \uplus \langle f, \mathcal{E}[\texttt{atomic } e] \rangle, \theta \rangle \rightarrow_{\mathrm{tf}} \langle \mathrm{T} \cup \langle f, \mathcal{E}[v] \rangle, \theta :: \tau' \rangle}$$

Transactional Transitions

$$\text{congruence}||_{\mathrm{tf}} \quad \frac{\langle \sigma, \tau, e \rangle \Rightarrow_\mathrm{t} \langle \sigma, \tau', e' \rangle}{\mathrm{T_x} \uplus \langle f, \sigma, \tau, \mathrm{F_s}, \mathrm{F_j}, \mathcal{E}[e] \rangle \Rightarrow_{\mathrm{tf}} \mathrm{T_x} \cup \langle f, \sigma, \tau', \mathrm{F_s}, \mathrm{F_j}, \mathcal{E}[e'] \rangle}$$

$$\text{fork}||_{\mathrm{tf}} \quad \frac{f' \text{ fresh}}{\begin{array}{l} \mathrm{T_x} \uplus \langle f, \sigma, \tau, \mathrm{F_s}, \mathrm{F_j}, \mathcal{E}[\texttt{fork } e] \rangle \Rightarrow_{\mathrm{tf}} \\ \mathrm{T_x} \cup \langle f, \sigma, \tau, \mathrm{F_s} \cup \{f'\}, \mathrm{F_j}, \mathcal{E}[f'] \rangle \cup \langle f', \sigma :: \tau, \varnothing, \varnothing, \mathrm{F_j}, e \rangle \end{array}}$$

$$\text{join}_1||_{\mathrm{tf}} \quad \frac{f' \notin \mathrm{F_j} \qquad \langle f', \sigma', \tau', \mathrm{F'_s}, \mathrm{F'_j}, v \rangle \in \mathrm{T_x} \qquad \mathrm{F'_s} \subseteq \mathrm{F'_j}}{\mathrm{T_x} \uplus \langle f, \sigma, \tau, \mathrm{F_s}, \mathrm{F_j}, \mathcal{E}[\texttt{join } f'] \rangle \Rightarrow_{\mathrm{tf}} \mathrm{T_x} \cup \langle f, \sigma, \tau :: \tau', \mathrm{F_s}, \mathrm{F_j} \cup \mathrm{F'_j} \cup \{f'\}, \mathcal{E}[v] \rangle}$$

$$\text{join}_2||_{\mathrm{tf}} \quad \frac{f' \in \mathrm{F_j} \qquad \langle f', \sigma', \tau', \mathrm{F'_s}, \mathrm{F'_j}, v \rangle \in \mathrm{T_x}}{\mathrm{T_x} \uplus \langle f, \sigma, \tau, \mathrm{F_s}, \mathrm{F_j}, \mathcal{E}[\texttt{join } f'] \rangle \Rightarrow_{\mathrm{tf}} \mathrm{T_x} \cup \langle f, \sigma, \tau, \mathrm{F_s}, \mathrm{F_j}, \mathcal{E}[v] \rangle}$$

Figure 5 Operational semantics of transactional tasks.

A join can only occur once f' has been resolved, i.e. once the task has been reduced to a single value v. Moreover, we require that the joined task has itself joined all of its children ($\mathrm{F'_s} \subseteq \mathrm{F'_j}$). This rule applies recursively: each of the tasks in $\mathrm{F'_s}$ should also have joined its children. Thus, this condition ensures that all effects of the task and all its descendants have been applied to its local store τ'.

On the first join of a task, its changes are pulled into the joining task, by merging its τ' into the local τ and adding its set of joined tasks $\mathrm{F'_j}$ to the local $\mathrm{F_j}$. On subsequent joins, by any task that has directly or indirectly joined f' before, the changes are not merged again, as these effects have already been merged. In both cases join resolves the future to its value v.

A task does not necessarily need to be joined by its parent: we do not require $f' \in \mathrm{F_s}$. Any task that is able to obtain a reference to the task's future can join it, using the same rules. This maintains the flexibility of traditional futures. However, a transactional task cannot be joined outside the transaction it was created in, as it depends on the initial state of the heap at the start of the transaction. This is enforced as the joined task should be in $\mathrm{T_x}$, and there is a separate such set for each transaction.

Conflict resolution function For simplicity, we did not describe the functionality provided by the custom conflict resolution function of Section 4.2 in the operational semantics. Instead, the child's value always overwrites the parent's value. To add this function, the operation $\tau :: \tau'$ in rule $\text{join}_1||_{\mathrm{tf}}$ should be replaced with an operation that calls the

```
(atomic
  (let [a (fork (let [d (fork …)
                      e (fork …)] …))
        b (fork (let [f (fork …)] …))
        c (fork (let [g (fork …)
                      h (fork …)] …))]
    …))
```

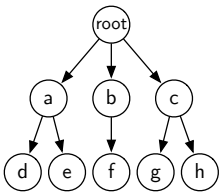


Figure 6 The tasks created in a transaction form a tree. By writing the tree in post-order notation, `d-e-a-f-b-g-h-c-root`, we see that task `b` can join tasks `d`, `e`, `a`, and `f`. It may be less obvious that task `f` can also join task `d`: task `a` may return `d`, and `f` can access `a`.

resolve function for conflicting writes in τ and τ'. The operations $\theta :: \tau$ in atomic$||_{\text{tf}}$ and $\sigma :: \tau$ in fork$||_{\text{tf}}$ do not need to be replaced: they represent non-conflicting writes.

5.2 Properties

The following properties are a result of the operational semantics:

Serializability This semantics is trivially serializable: at most one transaction is active at a time, as it is a high-level semantics, and is committed at once in atomic$|_{\text{tf}}$. The semantics thus describes how a correct implementation of transactional tasks should behave for the programmer, and forms a correctness criterion for more complex implementations. An actual implementation that allows multiple transactions to execute concurrently is discussed in Section 6.

Deadlock freedom The tasks created in a transaction form a tree, as illustrated in Figure 6, with the root task at the root of the tree. Each task is identified by a future, which is a reference that can be passed around and supports one operation: `join`. Each task can obtain the future of 1) its child tasks, as it created those; 2) its descendants, if a child returns the future of a descendant (e.g. the root task can access `d` if `a` returns `d`); 3) its earlier siblings (e.g. `b` can access `a`); and 4) any descendants of its earlier siblings, if a sibling returns the future of one of its descendants (e.g. `f` can access `d` if `a` returned `d`). Tasks cannot obtain their own future. Writing the tree in post-order notation defines a strict total order on the tasks, in which each task can access the futures of the tasks that come before it. This is a consequence of the lexical scoping of the language. This means that there cannot be circular dependencies between futures, and therefore deadlocks are impossible.

Coordination: a transaction's tasks are *all* committed, atomically Rule atomic$|_{\text{tf}}$ requires that all tasks created by the root task have been joined before commit ($\text{F}_\text{s} \subseteq \text{F}_\text{j}$). Furthermore, rule join$_1||_{\text{tf}}$ specifies that each task should join its sub-tasks before it in turn can be joined ($\text{F}'_\text{s} \subseteq \text{F}'_\text{j}$). As a result, all tasks created in the transaction should have been joined, directly or indirectly, into the root task before it can commit. If this is not the case, the program is incorrect. Consequently, the local store τ' of the root task contains the changes to the transactional state of *all* tasks in the transaction. In rule atomic$|_{\text{tf}}$, these changes are committed atomically.

In-transaction determinacy In a transaction, there is no non-determinism: given the initial state of the transactional heap θ, a transaction will always reduce to the same value v and local store τ', assuming that all conflict resolution functions are determinate. We say that a transaction is determinate. This can be proven using a technique similar to [10].

Listing 5 Overview of implementation of transactions and transactional tasks as an extension of Clojure. Abridged versions of the interfaces of the classes `Transaction` and `TxTask` are listed.

```
public class Transaction {
  // (dosync fn): runs fn in a transaction and returns the return value of fn.
  static public Object runInTx(Callable fn) {
    // If we're already in a transaction, use that one; else create one. In the new tx, fn is called, and an attempt to
    // commit is made. As long as committing fails, we rerun fn and retry.
  }
  void stop(); // Indicate that tx and its tasks should stop by setting stop flag.
}

// Implements Callable so that it can be created in a thread pool; implements Future so we can wait for its final value.
public class TxTask implements java.util.concurrent.Callable, java.util.concurrent.Future {
  static ThreadLocal<TxTask> task; // Transactional task running in current thread (can be null)
  Transaction tx;                  // Associated transaction

  Vals<Ref, Object> snapshot;    // In-tx values of refs on creation of this task (from all ancestors)
  Vals<Ref, Object> values;      // In-tx values of refs (set/commute)
  Set<Ref> sets;                 // Set refs
  Map<Ref, List<CFn>> commutes; // Commuted refs
  Set<Ref> ensures;              // Ensured refs
  Set<TxTask> spawned;           // Spawned tasks
  Set<TxTask> joined;            // Directly/Indirectly joined tasks

  // (fork fn): spawns a task. Creates TxTask if a transaction is running, or a regular Future otherwise.
  static public Future spawnFuture(Callable fn) { ... }
  // Create a transactional task in tx to execute fn. Snapshot is created from parent (= current thread).
  TxTask(Transaction tx, TxTask parent, Callable fn) { ... }

  Object doGet(Ref r) { ... }                          // (deref r)
  Object doSet(Ref r, Object v) { ... }                // (ref-set r v)
  Object doCommute(Ref r, IFn fn, ISeq args) { ... } // (commute r fn args)
  void doEnsure(Ref r) { ... }                         // (ensure r)
  void join(TxTask child) { ... }                      // (join child) (new)
}
```

6 Implementation

We implemented transactional tasks as an extension of Clojure, a Lisp dialect implemented on top of the Java Virtual Machine.[7] In this section, we briefly describe the core concepts of Clojure's STM and our modifications to it. Listing 5 lists the interface of the `Transaction` and `TxTask` (transactional task) classes.

Clojure represents transactional variables as `Refs`. It uses Multi-Version Concurrency Control (MVCC) [6]: each ref contains a (limited) history of previous values and the time at which they were set. As a result, transactions can read an older version of a ref even after it has been overwritten, preventing conflicts. Only if no recent-enough value is available in the limited history, will the transaction abort and restart.

A transaction contains several data structures to store its modifications (together they correspond to the local store τ of the operational semantics): `sets`, `commutes`, and `ensured` for refs modified using `ref-set`, `commute`, and `ensure`. The latest value of each modified ref is stored in the mapping `values`. In traditional Clojure these data structures are stored in the `Transaction`. To support transactional tasks, we move them to the `TxTask` class, as each task should have its own local store of modifications. Additionally, the sets of `spawned` and `joined` tasks are stored in the current task as well, mirroring the operational semantics.

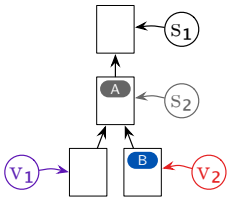
Clojure translates a transaction `(dosync e)` into `Transaction.runInTx(fn)`, where `fn` corresponds to a function executing `e`. The creation of a task using `(fork e)` is translated to `TxTask.spawnFuture(fn)` and returns a new `TxTask`, which runs in a new thread (using a thread pool). Upon its creation, the new task's `snapshot` and `values` are set to a copy

[7] It is available in the artifact, or at `https://github.com/jswalens/transactional-futures`.

(a) Code example.

```
(atomic
  1(ref-set gray A)
  2(fork 3(ref-set blue B)
         4(fork 5(ref-set green C))
         6(ref-set red D)
         ...)
  7(ref-set purple E)
  ...)
```

(b) Data after step 3.



(c) Data after step 7.

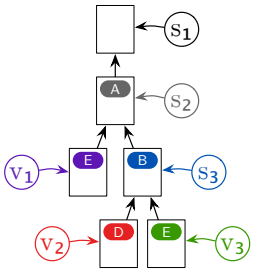


Figure 7 In the code example (a) three tasks are created. Each task contains a snapshot, which is immutable through the task's lifetime, and values, to which updated values for refs are written. (b) and (c) illustrate how the data structures are stored in memory.

of the parent's current `values`, as described in Section 6.1. The other data structures start empty. When reading a ref, `doGet(r)` searches for it first in the task's `values`, next in its `snapshot`, and finally in the ref's history. Modifying a ref calls `doSet(r, v)`, which adds the ref to `sets` and its value to `values`.

When a parent task joins a child, `(join child)` is translated to `parent.join(child)`, which takes several steps. Firstly, it waits until the child task has completed. Next, it checks whether the child task has joined all of its children. Then, the value of each ref in the `sets` of the child is copied from the child's to the parent's `values`, calling the resolution function on conflicts. The child's `sets`, `commutes`, `ensures`, and `joined` are appended to the parent's. Additionally, the child is added to its parent's `joined`. Lastly the child's final value is returned. On subsequent joins of the same child, only the final value is returned.

The transaction also contains a 'stop' flag. If one of the tasks encounters a conflict during its execution, it sets the transaction's stop flag and stops. The transactional operations (`doGet`, `doSet`, `join`...) contain a check point: they check the stop flag when they are called and abort their task if it is set. Consequently, when a task encounters a conflict and sets the stop flag, all other tasks in the same transaction will stop once they reach their next check point, and once all tasks have stopped the transaction is restarted.

When a transaction commits, at the end of `Transaction.runInTx(fn)`, it first checks whether the root task has joined all of its children: if it has we know all tasks have been joined (as explained in Section 5.2), otherwise an exception is thrown. Next, each modified ref is locked and the changes are committed. This may abort other transactions: by setting the stop flag of the other transaction, eventually all its tasks will stop and the conflicting transaction restarts.

6.1 Implementation of Snapshot and Local Store

Even though our implementation is a prototype, we briefly describe how the `snapshot` and `values` are stored, the most frequently used data structures in a transactional task. Even though the snapshot and values of a task are copied from its parent when it is created, we do not actually store duplicates.

Figure 7a lists a program that creates three tasks that each modify some refs. In Figure 7b we illustrate how the data structures are stored in memory after the third statement. We write s_i and v_i for the snapshots and values of task i. Each data structure comprises a linked list of hash maps. We exploit the fact that snapshots are immutable to share some of these hash maps between the data structures.

In the code example, task 1 (the root task) first sets A. A second task is forked and sets B. Consequently, after this step, s_1 is empty, v_1 and s_2 both contain A, and v_2 consists of A and B. Figure 7b illustrates how this is stored in memory. The snapshots are shared between the two tasks: these are immutable structures only used for look-up. The values of both tasks, v_1 and v_2, consist of a linked list that first contains a hash map that stores their private changes and next contains the shared snapshots. When ref B is updated in the second task, it is updated in the first hash map pointed to by v_2. When a ref is read in the second task, we iterate over the linked list pointed to by v_2, up the tree, until the ref is found.

Creating a new task, as in step 4, is now a matter of modifying some pointers. When task 3 is forked by task 2, the node that represented v_2 becomes the snapshot s_3 of the new task, with two empty children to contain the new values of tasks 2 and 3. After the last step, in Figure 7c, tasks 2 and 3 have updated their values with D and E respectively. Hence, v_3 now consists of the new values of task 3, the snapshot of task 3, the snapshot of task 2, and the snapshot of task 1.

By representing the `snapshot` and `values` data structures of a transactional task as a linked list of hash maps, the memory overhead of duplicated entries is eliminated. In exchange, the look-up time slightly increases as we need to iterate over the list of maps. The time to update a value is unchanged: a write happens directly in the first hash map. Forking is a matter of creating two new maps and adjusting an existing pointer. Joining still means copying values and potentially resolving conflicts, as explained earlier.

Furthermore, we performed another optimization for a common use case. In many programs it is common to create several tasks immediately after another, for example in a parallel map as in Section 3. This leads to a sequence of empty nodes in the tree. Instead of pointing a new child to an empty node, we directly point to the previous non-empty node. This optimization avoids the need to traverse empty nodes on look-ups. It is a safe optimization as non-leaf nodes in the tree are always snapshots and therefore immutable.

7 Case Studies and Experimental Results

We evaluate the applicability and performance benefits of transactional tasks using the STAMP benchmark suite [21]. STAMP consists of the eight applications listed in Table 1. It has been composed so as to represent a variety of use cases for transactions, from several application domains and with varying properties. We are interested in two characteristics in particular: (1) the transaction length, i.e. the average number of instructions per transaction in an execution; (2) time spent in transactions, i.e. the ratio of instructions in transactions over the total number of instructions in an execution of the program. When most of the execution time is spent in transactions, we can assume that those transactions execute performance-critical parts of the application. If these transactions are also long-running, they potentially benefit from more fine-grained parallelism.

As shown in Table 1 (page 8), five out of the eight applications in the STAMP suite spend a large proportion of time in transactions, three of which have long-running transactions: Labyrinth, Bayes, and Yada. In the rest of this section, we study Labyrinth and Bayes.

We also examined the Yada application. While its transactions have a relatively long execution time, they contain sequential dependencies that make it difficult to parallelize the different steps. Hence, we were unfortunately not able to achieve a speed-up using transactional tasks. The overhead of creating transactional tasks overshadows the benefit of performing only a small piece of the program in parallel. We therefore will not look at this application in further detail.

For these case studies, we first ported the STAMP applications from C to Clojure. Afterwards, we used transactional tasks to parallelize individual transactions where applicable.[8] In the performance results, the implementations with transactional tasks are compared with the original version in Clojure.

Experiments ran on a machine with two Intel Xeon E5520 processors, each containing four cores with a clock speed of 2.27 GHz and a last-level cache of 8 MB. HyperThreading was enabled, leading to a total of 16 logical threads. The machine has 8 GB of memory. Our transactional tasks are built as a fork of Clojure 1.6.0, running on the Java HotSpot 64-Bit Server VM (build 25.66-b17) for Java 1.8.0. Each experiment ran 30 times; the graphs report the median and the interquartile range.

7.1 Labyrinth

The Labyrinth benchmark was already introduced in Section 3. The goal of the benchmark is to connect given pairs of points in a grid using non-overlapping paths. For each pair, a breadth-first search is executed in a new transaction. In Section 4 we discussed how each iteration of the breadth-first search can process its elements in parallel using transactional tasks. Here, we optimized this solution to first distribute the elements into a configurable number of partitions, and then process these partitions in parallel. Furthermore, we ran the experiments on a three-dimensional grid of $64 \times 64 \times 3$ with 32 input pairs.

In the original version, there is one parameter t that influences the amount of parallelism: t worker threads will process input pairs in parallel. The maximal ideal speed-up in the original version is therefore t: in an ideal case where no transactions fail and the overhead is zero, we can expect a speed-up of maximally t. In the version that uses transactional tasks, another parameter p affects the parallelism: the number of partitions created on each iteration of the breadth-first search. Each of the t worker threads can create at most p partitions; therefore, the maximal number of threads and thus the maximal ideal speed-up in the version with parallel search is $t \times p$.[9]

In Figure 8 we measure the speed-up of running the program with several values of t and p. The speed-up is calculated relative to the version with sequential search and only one worker thread ($t = 1$). For the version that uses sequential search, the number of worker threads is increased (blue line). For the version with parallel search, both the number of partitions (different lines) and the number of worker threads (different points on the same line) are varied. The x axis denotes the maximal number of threads, i.e. t for the sequential search and $t \times p$ for the parallel search. In an ideal case, the measured speed-up would be equal to the maximal number of threads.

The blue line depicts the results of the original version of the Labyrinth application. Increasing the number of threads causes only a modest speed-up, because they find overlapping paths and consequently need to be rolled back and re-executed. This is shown in Figure 9, which lists the average number of attempts per transaction. If there is only one thread, each transaction executes only once, but as the number of threads increases each transaction re-executes several times on average. For 16 threads, the average transaction executes 2.10 times, i.e. it rolls back more than once. This hampers any potential speed-up.

In the version with parallel search, as the parameter p increases the speed-up improves, for small values of p. Each transaction now spawns p tasks, and consequently each transaction

[8] The code for these applications is available in the artifact, or at `https://github.com/jswalens/ecoop-2016-benchmarks`.

[9] To minimize the overhead of forking tasks, we ensure that each partition contains at least 20 elements.

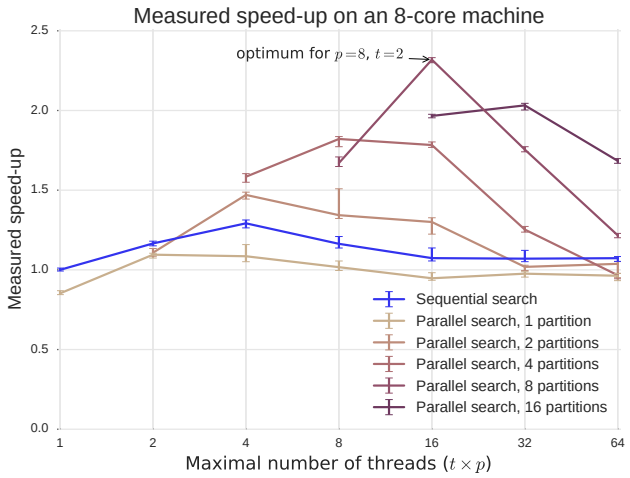


Figure 8 Measured speed-up of the Labyrinth application for the version with sequential search (blue line) and parallel search (other lines), as the total number of threads ($t \times p$) increases. Each point on the graphs is the median of 30 executions, the error bar depicts the interquartile range.

can finish its execution faster. On the tested hardware, an optimal speed-up of 2.32 is reached for $t = 2$ and $p = 8$, when two worker threads process elements and create up to eight partitions. For this case, the number of conflicts is low: each transaction executes 1.11 times on average (Figure 9).

Further increases in p lead to worse results: the additional parallelism does not offset the overhead of forking and joining tasks. Joining transactional tasks is expensive for this benchmark, as conflicts between the tasks are likely (two points expanding into a shared neighbor), and each conflict calls a conflict resolution function (the minimum, as explained in Section 4.2).

The parallel search with $p = 1$ (which does not actually search in parallel as only one partition is created) is slower than the sequential search. This is due to the different algorithms: the parallel algorithm creates several sets on each iteration to keep track of the work queue, while the sequential algorithm uses one list throughout.

These results demonstrate two benefits of transactional tasks. Firstly, the execution time of each transaction decreases by exploiting parallelism in the transaction. Secondly, the lower execution time of a transaction also means that the cost of conflicting transactions is decreased: each attempt takes less time. By varying the two parameters t and p, we can find an optimum between running several transactions simultaneously but risking conflicts (t) and speeding up the transactions internally but with more fine-grained parallelism (p).

Finally, to transform the original version with sequential search into the one with parallel search, out of 682 lines, 30 lines (4%) were removed and 78 lines (11%) were added. This corresponds to changing the sequential search algorithm into the parallel search algorithm, which is more complex.

7.2 Bayes

The Bayes application implements an algorithm that learns the structure of a Bayesian network given observed data [11, 21]. A Bayesian network consists of random variables and their conditional dependencies. Each variable in the Bayesian network is represented as a transactional variable that contains references to its parents and children. Initially, there

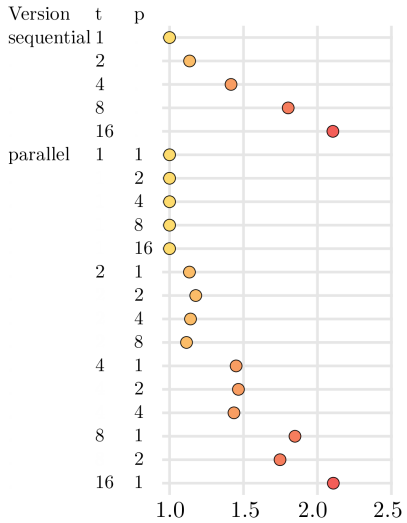


Figure 9 Average number of attempts per transaction for different values of t (transactions executing simultaneously) and p (partitions per transaction).

are no dependencies between the variables. A shared work queue contains the dependencies to insert next, and is initialized to one dependency per variable. t worker threads process the work queue in parallel: they insert the dependency into the network, and then calculate which dependencies (if any) could be inserted next, appending the best candidate to the work queue. The best candidate dependency is the one that maximizes a score function that calculates the capability of the network to estimate the input data. This is encapsulated in a transaction to prevent two dependencies from being added to the same variable simultaneously. Dependencies are inserted until the work queue is empty. As more dependencies are discovered, connected subgraphs of dependent variables form in the network.

Before the algorithm starts, the application generates the input data. Then, the t worker threads process the work queue in parallel. Figure 10 indicates that a typical execution spends 11.8% of its total time generating the input data, 88.1% learning the dependencies, and 0.1% validating the solution. We focus on the middle part only. In that part, 93.2% of the execution time is spent in the transaction that determines the best next dependency. The transaction contains a loop that calculates the score for each candidate and then selects the maximum. Each of the iterations of this loop is independent, and can therefore run in parallel using transactional tasks.

In Figure 11, we measure the speed-up of the learning phase as the number of worker threads (t) increases, for a network of 48 variables. The blue line is the original version: t threads process dependencies in parallel. The red line shows the version in which the loop is executed in parallel. Here, in each transaction, up to v transactional tasks run in parallel, where v is the number of Bayesian variables in the network (48 in our experiment). Therefore, the maximal ideal speed-up in the original version is t, while in the version with the parallel loop it is $t \times v$.

The speed-up of the original version (blue line) increases as number of threads increases, up to a speed-up of 2.75 for 16 threads. After this point, the speed-up plateaus. By examining the execution of the program, we find that even though a larger number of worker threads are created, only a limited number of them actually perform any work. The others are idle as not enough work is available after a certain point in the execution of the program.

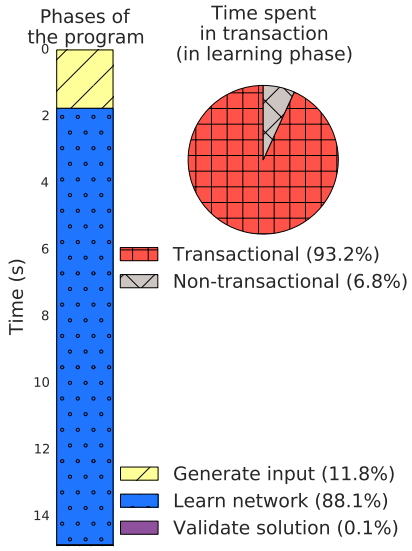


Figure 10 Proportion of time spent in different parts of the Bayes application (with $v = 48$).

In the version with parallel tasks, we see that even when there is only one worker thread processing one transaction at a time, the parallelization of its internal loop produces a speed-up of 2.88. By increasing the number of worker threads, a maximum speed-up of 3.45 can be produced for 5 worker threads. Again, the speed-up reaches a plateau as not enough work is available for all worker threads. However, the reached speed-up is higher than the original version as more fine-grained parallelism is available in each unit of work.

This result demonstrates another benefit of transactional tasks. In the original version, the amount of parallelism corresponded to the number of transactions, which is equal to number of work items. Hence, if at a certain point there are fewer work items than cores in the machine, not all potential parallelism is exploited. By introducing parallelism inside the transactions, we make better use of the available hardware. Even if there is limited work and therefore a limited number of transactions, transactional tasks allow us to make use of more fine-grained parallelism in the transactions.

Lastly, to modify the original version into the one that uses transactional tasks, only one line (out of 1248) had to be changed: the keyword `for` was replaced by `parallel-for`, a macro that uses transactional tasks internally to execute its iterations in parallel.

7.3 Conclusions

Based on these experiments, we draw the following conclusions:

- Out of the eight applications in the STAMP benchmark suite, which represents a variety of use cases for transactions, five spend a large proportion of their time in transactions and three of these have long-running transactions. We parallelized two of these three applications using more fine-grained transactional tasks.
- In the Labyrinth application, transactional tasks allow the breadth-first algorithm to be parallelized. This leads to a speed-up, by tuning the parameters for parallelism to have faster transactions and fewer conflicts.
- The Bayes application spends most of its execution time in a loop (in a transaction) that can be trivially parallelized. As there is only limited work available, at a certain point the

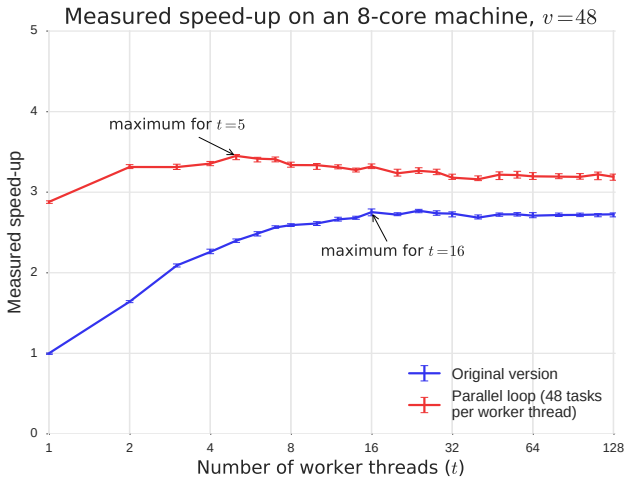


Figure 11 Measured speed-up of the learning phase for the Bayes application, as the number of threads increases. The blue line shows the original version. The red line shows the version with a parallel for loop, where each of the (at most) 48 iterations is executed in parallel.

number of transactions is lower than the number of cores in the machine. Transactional tasks allow us to introduce more fine-grained parallelism. This is a matter of changing `for` into `parallel-for`, and increases the maximal speed-up on an eight-core machine from 2.75 for the original version to 3.45 for the version with transactional tasks.

These results lead us to believe that transactional tasks allow developers to improve the performance of their transactional applications with only limited effort. Moreover, as our implementation is a relatively simple prototype in Clojure, further optimizations could decrease the overhead and improve its performance (e.g. for the Bayes application). In future work we would also like to explore the applicability of transactional tasks for other programs.

It should be possible to apply transactional tasks to other STM systems, such as Haskell or ScalaSTM. In those systems, the fact that the studied applications allow parallelism in the transaction also applies, the development effort to introduce them should be similar, but depending on the implementation the speed-up may be different.

8 Related Work

We will briefly discuss three categories of related work: nested and parallel transactions, concurrency models with deterministic access to shared memory, and work that allows parallel tasks to share memory.

Nested, multithreaded, and nested parallel transactions *Nested transactions* [23, 5, 24, 25] are subtransactions created in a transaction. Nested transactions attempt to commit separately from their parent. Hence, they can fail separately, thus requiring only a portion of the work to be rolled back. This can improve the performance of large transactions. In contrast to transactional tasks, nested transactions do not execute in parallel, as they correspond to nested `atomic` blocks and not the nesting of `fork` in `atomic`.

Haines et al. [15] and Moore and Grossman [22] allow threads to be created in a transaction, i.e. *multithreaded transactions*. However, there are no guarantees on the access to shared memory by threads within a transaction: they may read and modify shared transactional variables concurrently, thus permitting race conditions.

Transactional Featherweight Java [28] combines *nested and multithreaded transactions*: a transaction can spawn threads that contain nested transactions. When a nested transaction commits, its changes are written to its parent. Conflicts are explicitly forbidden: the value of a variable in a child must be the same as its value in the parent. *Nested parallel transactions* [1, 4, 2, 29] are similar, but resolve conflicts using the traditional serializability of transactions: the transaction that commits last wins. This is the main difference with transactional tasks, which resolve conflicts deterministically using conflict resolution functions, thus guaranteeing *in-transaction determinacy* (Sections 4.3 and 5.2). The second major difference is that nested parallel transactions roll back on conflicts between siblings, while transactional tasks do not, as they resolve the conflicts instead. In an application without such conflicts, both models operate equivalently. However, in our motivating example, the Labyrinth application, such conflicts frequently occur: the parallel expansion of the breadth-first search causes conflicts on overlapping cells. Nested parallel transactions would cause frequent rollbacks of the inner transactions, essentially sequentializing them, detrimental to performance. Transactional tasks run in parallel, but rely on the developer for an appropriate conflict resolution function. We expect both models to be suited for different applications. In applications in which non-top-level sibling transactions conflict, we expect transactional tasks to perform better.

Deterministic access to shared memory The model provided by transactional tasks *in* the transaction is similar to two existing concurrency models. Firstly, Concurrent Revisions are a model for task parallelism with shared memory [9]. Its concurrent tasks share memory using *versioned variables*. When a task is forked, a conceptual copy is made of the versioned variables; when a task is joined, the changed variables are merged into the joining task. This resembles how transactional variables behave *inside* a transaction in our model. However, between the transactions we provide serializability. As such, Concurrent Revisions provide determinacy for the complete program, while transactional tasks provide determinacy *in* the transactions, and serializability of the transactions as a whole.

A similar model is provided by Worlds [30], in which the program state is reified as a *world*. The world can be forked into a child world, a conceptual copy of all program state. The state in a child world can be updated, and eventually committed back (merged) into its parent world. As such, worlds also behave similarly to the transactional variables *in* a transaction. However, the Worlds model does not provide parallelism: a child world does not run in parallel with its parent. Instead, Worlds are used as a mechanism to 'undo' changes to the program state. As a result, when a child world is merged in its parent there will be no conflicts, as the parent has not changed in the mean time. In the case of transactional tasks and Concurrent Revisions, it is possible for the parent to have changed as well, and a form of conflict resolution between parent and child is needed.

Parallel tasks with shared memory Otello [34] allows parallel tasks to access shared memory, while still running the tasks in isolation. To this end, it introduces *assemblies*, which consist of a task and the set of shared objects it owns. When two assemblies conflict, one is re-executed after the other has finished. However, while Otello re-executes code, it does not provide transactions and as such does not guarantee serializability.

9 Conclusion

Many modern programming languages and frameworks support multiple concurrency models. However, the combination of these concurrency models is often either not supported or not

well defined. In existing languages, using futures to increase the parallelism within a single transaction is either not allowed (Haskell), or leads to unexpected behavior (Clojure, Scala).

This paper introduces transactional tasks as a mechanism to enable safe parallelism within transactions. Each transactional task executes within the context of its encapsulating transaction, hence, transactions remain serializable. Furthermore, the different tasks of a single transaction are coordinated: they are all committed or retried together. Lastly, transactional tasks do not introduce non-determinism, and behave as nested transactions, thereby providing a straightforward and familiar mental model to the programmer.

This paper provides a formalization of transactional tasks, and discusses its implementation on top of Clojure. Our approach is validated through a case study of several applications from the STAMP benchmark suite. We show that our approach allows finer-grained parallelism of performance-critical, long-running transactions to be exploited, leading to a higher speed-up. As a result, we believe that transactional tasks successfully combine futures and Software Transactional Memory, allowing the parallelism of a program to be fully exploited with limited developer effort, while preserving the properties of the separate models where possible.

Acknowledgements. We would like to thank Stefan Marr for insightful discussions and comments on early drafts. We also thank the anonymous reviewers for their feedback.

References

1 K. Agrawal, J. T. Fineman, and J. Sukha. Nested Parallelism in Transactional Memory. In *PPoPP*, 2008.

2 W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun. Implementing and evaluating nested parallel transactions in software transactional memory. In *SPAA*, 2010.

3 H. C. Baker and C. Hewitt. The incremental garbage collection of processes. In *Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, 1977.

4 J. Barreto, A. Dragojević, P. Ferreira, R. Guerraoui, and M. Kapałka. Leveraging parallel nesting in transactional memory. In *PPoPP*, 2010.

5 C. Beeri, P. A. Bernstein, and N. Goodman. A model for concurrency in nested transactions systems. *Journal of the ACM*, 36(2):230–269, 1989.

6 P. A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

7 G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1999.

8 R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPoPP*, 1995.

9 S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *OOPSLA*, 2010.

10 S. Burckhardt and D. Leijen. Semantics of Concurrent Revisions. In *ESOP*, 2011.

11 D. M. Chickering, D. Heckerman, and C. Meek. A Bayesian approach to learning Bayesian networks with local structure. In *UAI*, 1997.

12 M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.

13 C. Flanagan and M. Felleisen. The Semantics of Future and Its Use in Program Optimizations. In *POPL*, 1995.

14 R. Guerraoui, M. Kapałka, and J. Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *European Conference on Computer Systems*, 2007.

15 N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems*, 16(6):1719–1736, 1994.

16 R. H. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

17 T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP*, 2005.

18 M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. *ACM SIGARCH Computer Architecture News*, 21:289–300, 1993.

19 S. Imam and V. Sarkar. Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. In *ECOOP*, 2014.

20 C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, 1961.

21 C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Symposium on Workload Characterization*, 2008.

22 K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL*, 2008.

23 J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

24 J. E. B. Moss and A. L. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, 2006.

25 Y. Ni, V. S. Menon, A. 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*, 2007.

26 N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

27 S. Tasharofi, P. Dinges, and R. E. Johnson. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models? In *ECOOP*, 2013.

28 J. Vitek, S. Jagannathan, A. Welc, and A. L. Hosking. A Semantic Framework for Designer Transactions. In *ESOP*, 2004.

29 H. Volos, A. Welc, A. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy. NePaLTM: Design and Implementation of Nested Parallelism for Transactional Memory Systems. In *ECOOP*, 2009.

30 A. Warth, Y. Ohshima, T. Kaehler, and A. Kay. Worlds: Controlling the Scope of Side Effects. In *ECOOP*, 2011.

31 I. Watson, C. Kirkham, and M. Lujan. A Study of a Transactional Parallel Routing Algorithm. In *PACT*, 2007.

32 A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *OOPSLA*, 2005.

33 Y. Zhang and E. A. Hansen. Parallel Breadth-First Heuristic Search on a Shared-Memory Architecture. In *Heuristic Search, Memory-Based Heuristics and Their Applications*, 2006.

34 J. Zhao, R. Lublinerman, Z. Budimlić, S. Chaudhuri, and V. Sarkar. Isolation for nested task parallelism. In *OOPSLA*, 2013.

A Syntactical and Semantical Differences with Clojure and Haskell

The syntax and operational semantics introduced in Section 2 closely matches Clojure and Haskell, with only a few insignificant differences. These are detailed in this appendix.

A.1 Clojure

Clojure 1.7.0 (released June 30, 2015) differs from the presented syntax in the following ways:

- Clojure encapsulates all forms in parentheses, as S-expressions. Furthermore, it has a slightly different syntax for `let`.
- `fork` and `join` are named `future` and `deref` respectively. That is, `deref` is overloaded for both futures and transactional variables.
- Clojure's (`dosync` $\overline{e}$) is equivalent to our `atomic` (`do` $\overline{e;}$).

The semantics differ only on these two points:

- Clojure allows `ref` and `deref` to be used outside a transaction: Clojure's (`ref` e) and (`deref` e) are equivalent to our `atomic` (`ref` e) and `atomic` (`deref` e), but with an optimized implementation.
- Clojure supports `alter`, `commute`, and `ensure`, which are essentially variations of `ref-set` with different performance characteristics.

Finally, Clojure allows futures to be created in a transaction, as described in Section 3.

A.2 Haskell

Syntactically, Haskell writes `forkIO`, `atomically`, `newTVar`, `readTVar`, and `writeTVar` for `fork`, `atomic`, `ref`, `deref`, and `ref-set`. The semantics differ in the following ways:

- Haskell's `forkIO` returns a thread identifier, and not a future. Nevertheless, our formalization models the use of transactions in tasks, in Haskell one uses `atomically` in a task created using `forkIO`.
- Moreover, Haskell does not support the `join` operation on thread identifiers. Instead, waiting for a thread and retrieving its result is usually implemented manually using an `MVar`[10]; while our language uses futures for this purpose. This aspect does not affect the problem statement of Section 3, but porting our solution to Haskell does require adding a `join` operation to Haskell.
- Transactions are encapsulated in the `STM` monad, and the main program is encapsulated in the `IO` monad. This leads to a different semantics of the `do` block, which in Haskell is syntactic sugar for monad sequencing. Haskell's do notation allows monadic binding (`<-`) and `let` binding, and may require `return`.
- Haskell does not allow multiple `atomically` blocks to be nested: the type signature of `atomically` is `STM a -> IO a`, and a result of type `IO a` cannot be used in an `STM` block.
- Haskell supports `retry` to abort a transaction, and `orElse` to compose two alternative `STM` actions.

Lastly, one could see transactional tasks as the addition of `forkSTM` to Haskell, extending its `forkIO` to usage in the STM monad.

[10] As indicated in the documentation of the Control.Concurrent package at `https://hackage.haskell.org/package/base-4.8.1.0/docs/Control-Concurrent.html#g:12`.

Staccato: A Bug Finder for Dynamic Configuration Updates*

John Toman[1] and Dan Grossman[2]

1 **Department of Computer Science & Engineering**
University of Washington
Seattle, WA, USA
jtoman@cs.washington.edu

2 **Department of Computer Science & Engineering**
University of Washington
Seattle, WA, USA
djg@cs.washington.edu

Abstract

Many modern software applications are highly configurable, allowing configuration options to be changed even during program execution. When dynamic configuration updating is implemented incorrectly, program errors can result. These program errors occur primarily when stale data—computed from old configurations—or inconsistent data—computed from different configurations—are used. We introduce Staccato, the first tool designed to detect these errors. Staccato uses a dynamic analysis in the style of taint analysis to find the use of stale or inconsistent configuration data in Java programs. It supports concurrent programs on commodity JVMs. In some cases, Staccato can provide automatic bug *avoidance* and semi-automatic repair when errors occur.

We evaluated Staccato on three open-source applications that support complex reconfigurability. Staccato found multiple errors in all of them. Staccato requires only modest annotation overhead and has moderate performance overhead.

1998 ACM Subject Classification D.2.9 Software Configuration Management

Keywords and phrases Dynamic Configuration Updates, Dynamic Analysis, Software configuration

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.24

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.2.1.14

1 Introduction

Today's software is highly configurable [42, 43, 17]. This configurability increases the reusability and portability of software. Software configurations can be specified in several different ways: static compile-time configuration options [20], command-line options that can be changed on each program run [37], or options specified in a configuration file read at startup. Along with high levels of configurability, software is increasingly subject to stringent uptime requirements; restarting an application to effect a configuration change is not

* This paper is based upon work supported in part by the National Science Foundation under Grant No. CCF-1064497AM001, and upon research sponsored in part by DARPA under agreement number FA8750-16-2-0032.

always feasible. To accommodate configurability and robustness, many applications support configuration options that can be changed at runtime. We call a configuration change at runtime a *dynamic configuration update* (DCU).

Unfortunately, it is difficult to implement dynamic configuration updates correctly. A bug[1] in Solr,[2] an open-source text search and indexing server, demonstrates this. The user may specify *analyzers* that process text before it is indexed. These analyzers apply case normalization, remove stop words, and so on. The user specifies analyzers in a configuration file read by Solr at startup. Solr also provides a reload command which re-reads the configuration file and re-initializes the system. However, the analyzers were not updated to reflect the new configuration, even though Solr reported the reload complete. Solr would then silently misprocess data and incorrectly answer user queries. The only indication that something was wrong was Solr's output, which required careful inspection on the part of the user. We found similar defects in multiple applications.

This paper presents, to the best of our knowledge, the first study of defects in dynamic configuration update implementations. We have developed a dynamic analysis that can assist developers in finding and diagnosing defects in their DCU implementations. We implemented our technique in Staccato (STAle Configuration and Consistency Analysis Tool), a prototype DCU error detection tool for Java programs.

Our approach checks one of two alternative correctness conditions for DCU systems, as chosen by the programmer. The first condition states that old versions of configuration options may not be used after a configuration update. The second states that only one version of a configuration option may be used during a single method execution. Both conditions provide a possible specification of program behavior in the presence of dynamic configuration updates. Choosing the appropriate correctness condition for a program unit requires domain knowledge. We have the user of Staccato select the appropriate condition with a few lines of high-level annotations (fewer than 0.7 per 1,000 SLOC in our evaluation).

Underlying both conditions is the notion of *versioning* the software configuration. For every program value, Staccato tracks which configuration options, and what versions of those options, were used to construct that value. This information moves with a value as it transformed and combined with other values. Whenever a value is read by the program, Staccato checks that the value does not violate the correctness condition selected by the programmer. Our analysis supports concurrency and requires no changes to the JVM.

In addition to bug finding, Staccato provides support for program repair and bug avoidance. Staccato can transparently repair program schedules to hide insufficient synchronization around configuration accesses. In addition, when Staccato detects a value built from an out-of-date configuration, it calls a programmer-provided callback to update the stale value. We used this functionality to repair DCU errors that we discovered in our evaluation. We also found this technique to be effective enough that for some projects, we were able to add DCU support for options that previously required a restart.

We applied Staccato to three large open-source projects with extensive support for dynamic configurability. These were code-bases with which we were not previously familiar, but we were still able to effectively use Staccato to find DCU defects in all of them.

In summary, this paper makes the following contributions:

- We provide two correctness conditions for software that supports dynamic configuration updates (DCU).

[1] `https://issues.apache.org/jira/browse/SOLR-3587`
[2] `http://lucene.apache.org/solr/`

```
class RequestManager implements Reloadable {
  String targetIp, apiKey;
  RequestManager() {
     targetIp = Config.get("request.target-ip");
     apiKey = Config.get("request.api-key");
  }
  String doRequest() {
     Request req = new ApiRequest(targetIp);
     req.send(apiKey);
     return req.response();
  }
  void reloadConfig() {
     targetIp = Config.get("request.target-ip");
  }
}
```

Figure 1 A bug caused by an incomplete configuration update: after an update `reloadConfig()` does not read the updated `"request.api-key"` option.

```
class SpotService implements WebService {
  String handleSpotRequest() {
    return Config.get("verb") + " Spot, " + Config.get("verb") + "!";
  }

  void handleUpdateRequest(String newVerb) {
    Config.set("verb", newVerb);
  }
}
```

Figure 2 A bug caused by an inconsistent view of the configuration: the `handleSpotRequest` method can observe two versions of the `"verb"` option in one execution.

- We define the problem of detecting and diagnosing errors in DCU code.
- We describe how errors in DCU implementations can be discovered with a novel dynamic information-flow tracking approach.
- We describe Staccato, a tool implementing this approach for Java programs.[3]
- Using Staccato, we show that bugs in DCU implementations affect multiple open-source projects, and that our technique is effective for finding these errors.

The rest of the paper is organized as follows. Section 2 introduces and justifies the correctness conditions checked by our approach. Section 3 introduces an information-flow tracking technique for detecting correctness violations in programs that support DCU. Section 4 details our approach for bug avoidance and program repair. Section 5 describes our implementation. Section 6 presents the results from applying Staccato to three open-source software projects. Section 7 covers related work. Section 8 concludes.

2 Correctness Conditions for Dynamic Configuration Updates

Staccato identifies errors caused by defects in dynamic configuration update mechanisms. We are unaware of any agreed upon correctness definition for DCU implementations either

[3] Staccato is open-source. Our implementation and the tests we ran for this paper can be found at `https://github.com/uwplse/staccato`

in industry or the research literature (see Section 7). Accordingly, we first give two correctness definitions for DCU schemes that we developed after surveying configurable software systems. Both conditions provide a specification for program behavior in the presence of DCU. The programmer selects which condition to use for an application, with the option of switching conditions (or opting out altogether) on a per-method or per-class basis with annotations.

We begin with two motivating examples found in Figures 1 and 2. Both examples are invented code fragments that are representative of two common errors we found during our survey of configurable software. The `Config` class provides a mapping of configuration options to string values. In Figure 1, the user may update either the target IP or API key configuration options, after which the `reloadConfig()` method is called. The implementation has an error in its handling of the update: the `reloadConfig()` method fails to read the new value of the `"request.api-key"` configuration option. As a result, the configuration update requested by the user is not completely applied. The Solr bug described in the introduction is similar in form to this example.

Figure 2 shows a consistency issue that we frequently found in configurable software. In this example, `SpotService` implements a web service that can be configured with an administrative command. The primary functionality, expressed in the `handleSpotRequest` method, is to simply echo back sentences of the form, `"Run Spot, Run!"`. The verb in the response is controlled with the `"verb"` configuration option. An administrator can update the `"verb"` option by sending an update request that is handled by the `handleUpdateRequest` method. If an update request is received concurrently with a regular client request, a user can receive unexpected responses such as `"Run Spot, Bark!"` This error occurs because the use of `"verb"` in `handleSpotRequest` is not *atomic*: a single invocation of `handleSpotRequest` can observe two inconsistent versions of the `"verb"` configuration option.

Guided by these two examples and many others like them found in real programs, we have developed two conditions for software that supports DCU. The first condition is as follows:

▶ Correctness Condition 1 (Staleness). An execution must observe only values derived from the most *up-to-date* version of the program configuration.

We call correctness condition C1 the *staleness* condition as it states that a program may not use values built from stale configurations. This correctness condition is potentially violated in both examples. In the first example, after a configuration update the program will read a stale version of the `"request.api-key"` option on line 9 in Figure 1. In the second example, the string returned by `handleSpotRequest` may be derived (partially or completely) from an old version of the `"verb"` option.

While condition C1 is sufficient to identify the bug in Figure 2, it is overly strict. For example, consider again the scenario where the service in Figure 2 receives two concurrent requests handled by `handleSpotRequest` and `handleUpdateRequest` respectively. Due to nondeterminism in thread scheduling, `handleSpotRequest` may return the response `"Run Spot, Run!"` after `handleUpdateRequest` has updated `"verb"` to another value, such as `"Bark"`. This outcome violates condition C1, but some systems depend on allowing this behavior for high performance. A more desirable correctness condition would rule out only *inconsistent* outcomes, such as `"Run Spot, Bark!"`, which violate the programmer's expectations about configuration behavior. This observation motivates a second correctness condition:

▶ Correctness Condition 2 (Consistency). A method execution may observe only a *single* version of each configuration option.

```
1 String foo(String a, String b, boolean t) {
2   if(t) { return a; } else { return b; }
3 }
4 // V["foo"] = 2
5 String s = Config.get("foo"); // s^H = {"foo" → 2}
6 Config.set("foo", "..."); // V["foo"] = 3
7 String r = Config.get("foo"); // r^H = {"foo" → 3}
8 String q = foo(r, s, false); // q^H = {"foo" → 2}
9 q = foo(r, s, true); // q^H = {"foo" → 3}
```

Figure 3 Example configuration histories in a program. The history of the configuration value returned by `Config.get` on line 5 maps `"foo"` to the current version of `"foo"`, $\mathcal{V}$["foo"] = 2. After the update on line 6, the epoch of `"foo"` is incremented and the value returned on line 7 is tagged with the new version. Notice that on lines 8 and 9 the configuration histories of `r` and `p` have moved through method parameters and return statements automatically.

In other words, a program may use an old version of a configuration option provided that version is not mixed with any other versions of the same option.

Both correctness conditions are useful in different situations. Which condition is appropriate for a given program is application dependent. In practice, we found that condition C2 (consistency) was a reasonable default for the programs in our evaluation set. Nevertheless, the staleness condition is a good fit for global, configurable objects, such as caches, database connections, etc. This led to a useful rule of thumb for using our technique: use the staleness condition for methods that manipulate configurable objects stored in static class members and use the consistency condition everywhere else. This was sufficient to find several bugs, and required fewer than 1 annotation per 1,000 SLOC across our evaluation set.

3 Technique

StaCCato detects errors in DCU schemes with a dynamic information-flow analysis in the style of taint analysis. There are three pieces of StaCCato's analysis, described in the next three subsections. First, StaCCato versions the software configuration and associates each program value with a *configuration history* (Section 3.1). A value's history records which options, and what *versions* of those option, were used to build that value. Second, when values are combined, their configuration histories are merged to produce a history for the output value (Section 3.2). This *propagation* models the flow of configuration information through a program. Finally, whenever a value is read by the program, StaCCato checks for violations of the DCU correctness conditions (Section 3.3). To ease explication, we will first describe our technique in terms of only detecting violations of the staleness condition (C1). Our approach for the consistency condition (C2) is presented in Section 3.4 as an extension.

3.1 Configurations Values

StaCCato models the configuration of a program as a global map from strings to strings. Following standard terminology, we call each key an *option*. We call each value in the map a configuration *value*. StaCCato also internally associates each configuration option with an *epoch* counter. These option epochs *version* the configuration values of a program. The epoch counters are stored in a map $\mathcal{V}$ from options to epochs: the current epoch of an option is denoted $\mathcal{V}[o]$. When an option o is updated, the value of $\mathcal{V}[o]$ is incremented by one. We will use e to denote arbitrary epoch values.

```
1 // V["foo"] = 2 and V["bar"] = 4
2 String f = Config.get("foo"); // f^H = {"foo" → 2}
3 String g = Config.get("bar"); // g^H = {"bar" → 4}
4 String h = f + g; // h^H = {"foo" → 2, "bar" → 4}
5 Config.set("foo", "..."); // V["foo"] = 3
6 String j = h + Config.get("foo"); // j^H = {"foo" → 2, "bar" → 4}
```

Figure 4 An example of history propagation and the history merge operation. On line 4, the input configuration histories have disjoint domains and therefore the merge operation is trivial. On line 6 the input configuration histories both have `"foo"` in their domains. According to the merge rule, the smaller of the two epochs for `"foo"` is chosen, which in this case is 2.

Staccato also tags each program value with a *configuration history*. The configuration history for a value v records the configuration options used to construct that value, along with the epochs (i.e., versions) of those options. The notation $v^{\mathcal{H}} = \{o_1 \to e_1, o_2 \to e_2, \ldots\}$ denotes that v was constructed using the value of configuration options o_1, o_2, ... at versions e_1, e_2, ... We will use $v^{\mathcal{H}}$ as a function mapping strings (option names) to epochs. $dom(v^{\mathcal{H}})$ denotes the set of options that appear in the configuration set $v^{\mathcal{H}}$. The notation $x^{\mathcal{H}}$ may be used when x is a variable to denote the history of the value referenced by x.

Configuration histories are associated with program values (not textual program variables). Thus, as a value moves through field array, variable assignments, or method boundaries, its configuration history automatically moves with it. This is important as DCU bugs involve complex information-flow: the Solr bug from the introduction involved 7 classes.

Configuration histories are empty for most values: only values that depend on the program configuration may have non-empty histories. Initially, the only values associated with non-empty configuration histories are configuration values themselves. When an option o is retrieved from the program's configuration, the returned configuration value is tagged with the history $\{o \to \mathcal{V}[o]\}$ (recall that $\mathcal{V}[o]$ is the current epoch of o). An example of this tagging and epoch counter updating can be found in Figure 3.

3.2 Propagation

Staccato tracks configuration values as they are combined with other values or transformed by the program. Thus, when multiple values are combined to create a new value, Staccato ensures that the new value's configuration history soundly captures all configuration information associated with the source values. The process of transferring configuration histories from operands to outputs is called *propagation*.

During propagation, configuration histories are merged together pairwise to yield the final configuration history of the output value. The merge operation for configuration histories is denoted $v^{\mathcal{H}} \cup u^{\mathcal{H}}$. If $dom(v^{\mathcal{H}}) \cap dom(u^{\mathcal{H}}) = \emptyset$, then the merge operation is a simple union. However, it is possible that some option o appears in both $v^{\mathcal{H}}$ and $u^{\mathcal{H}}$. Staccato conservatively handles this situation. Suppose that $v^{\mathcal{H}}[o] = e$ and $u^{\mathcal{H}}[o] = e'$. The output configuration history maps o to the epoch $min(e, e')$. This definition ensures that the epoch recorded for a configuration option o in a configuration history $x^{\mathcal{H}}$ is the oldest version of o used (either transitively or directly) to construct x. Figure 4 demonstrates history propagation and an example of the merge operation.

Staccato does not check for violations of the correctness condition during propagation. Recall that Staccato allows the programmer to select one of two possible correctness conditions or to opt out of checking altogether. If we integrated checking into propagation, we would have to implement three different propagation operations: one for each correctness

```
1 class DBConnection {
2   @StaccatoPropagate(RETURN)
3   static DBConnection connect(String host, String user) { ... }
4   String fetchRow(String query) { ... }
5 }
6 User getUser(...) {
7   String host = Config.get("db-host"), user = Config.get("db-user");
8   DBConnection conn = DBConnection.connect(host, user);
9   String userInfo = conn.fetchRow(...);
10  return new User(userInfo);
11 }
```

Figure 5 Propagation involving objects dependencies. The `@StaccatoPropagate` annotation the `connect` method propagates the configuration histories of the `host` and `user` parameters to the `DBConnection` object returned from the method. As a result, the configuration history of the `conn` in the `getUser` method has the `"db-host"` and `"db-user"` options in its configuration history.

condition and one for propagation without any checking. By separating the check and propagation operations, we can apply a uniform propagation operation across the entire program and vary the check operation depending on the correctness condition selected by the programmer.

Propagation for Object Types

A key design decision we faced is where propagation should occur. Operations such as integer arithmetic, boolean operations, and string concatenation all naturally propagate configuration information from source operands to output values automatically. Although sufficient for tracking simple dependencies, propagation involving only primitive values will miss dependencies at the object level. Consider the example in Figure 5. In order to verify that the usage of the connection object created on line 8 is correct, `conn` must inherit the configuration information of the `host` and `user` configuration values. Therefore, Staccato also supports propagation from method arguments to outputs. This method-level propagation expresses configuration dependencies at a higher level of abstraction than what is possible with just primitive operations.

The user opts-in to method-level propagation on a per-method basis with annotations like the one seen on line 2 in Figure 5. The annotation argument indicates which output of the method is the target of propagation. There are two possible outputs: the return value (specified with the `RETURN` argument seen in the example) or the method receiver. The receiver output is used primarily for modeling side-effecting methods such as setters or constructors. After an annotated method is executed, Staccato combines the configuration histories of the method's arguments with the existing history of the receiver or return value as appropriate. For the purpose of propagation to the return value, the method receiver (if applicable) is treated as an argument. In the database example, the propagation annotation ensures that $\texttt{conn}^{\mathcal{H}}$ contains the `"db-user"` and `"db-host"` options.

We experimented with an eager approach that propagated configuration histories unconditionally from method arguments to method receiver/return values. This seemed reasonable, as function outputs generally depend on input values. This strategy *would* properly capture the relationship between the `"db-host"` and `"db-user"` options and the `conn` object without user annotation. In practice, this approach suffers from a significant loss of precision. In Figure 5, automatic propagation also "taints" the value read from the database on line 9 with configuration information. By extension, the `User` object created on line 10 would also contain the database configuration options in its configuration history. However,

using a `User` object built with data read from an old connection is arguably *not* an error. This scenario is the expected behavior envisioned when a configurable database connection was implemented. In general, automatic propagation tags program data with unintuitive configuration histories, leading to false positives.

In our experience, configuration propagation is fairly rare; compared to the number of method definitions in our evaluation code-bases, the number of propagation annotations required was quite low, requiring fewer than 30 annotations across the entire evaluation set. We also found that propagation annotations need only be applied at natural API method boundaries, requiring no reasoning about whole program flow or API usage sites.

3.3 Checking

The final component of Staccato checks that the program does not observe values constructed from an out-of-date version of the configuration. Our approach actually checks an equivalent condition: that the value read by the program reflects only the most recent version of the configuration. A value v is up-to-date with respect to the current version of the configuration (as recorded in $\mathcal{V}$) iff: $\forall o \in dom(v^{\mathcal{H}}), v^{\mathcal{H}}[o] = \mathcal{V}[o]$. In other words, only the most recent configuration values may have been used (transitively or directly) to construct v. This check is performed on value reads. For the purposes of our analysis a value is read when it is: 1) passed as an argument to a method, 2) read from a field, 3) returned from a method, 4) read from a local variable, or 5) read from an array. If Staccato detects that the condition has been violated, we have identified a DCU defect and alert the user.

3.4 Extensions for Checking Consistency

We now discuss how to extend our approach to check for violations of the consistency condition (C2). Recall that the second correctness condition states that a method execution may not observe inconsistent versions of the program configuration. Checking this condition requires extending the representation of configuration histories and the merge operator.

In addition to option epochs, Staccato also tracks an extra bit for each option o in a configuration history. This bit is called the consistency flag. We denote this epoch-bit pair with $\langle e, f \rangle$, where e is an epoch and f is the new consistency flag. The consistency flag becomes set when Staccato detects that different versions of the same option have been used to construct a value. Any values derived (in terms of the propagation described in Section 3.2) from an inconsistent value are themselves marked as inconsistent. The consistency flag is initially unset for configuration values returned from the program's configuration: by definition, a configuration value always represents a single version of a configuration option.

The definition of the consistency flag gives rise to the following extension to the merge operator. Given two configuration histories to be merged, $u^{\mathcal{H}}$ and $v^{\mathcal{H}}$, such that $u^{\mathcal{H}}[o] = \langle e, f \rangle$ and $v^{\mathcal{H}}[o] = \langle e', f' \rangle$, the consistency flag in the merged configuration history is set iff: $e \neq e' \vee f = 1 \vee f' = 1$. The epoch for o is computed using the *min* function as previously described.

Given these extensions to configuration histories and the merge operation, checking the consistency condition is straightforward. The consistency condition requires that at most one version of each option may be used to build a value. This is precisely what the consistency flag tracks. Thus, checking a value v against the consistency condition entails checking that all consistency flags in $v^{\mathcal{H}}$ are unset. That is, a value v reflects a consistent view of the configuration iff: $\forall o \in dom(v^{\mathcal{H}}).v^{\mathcal{H}}[o] = \langle e, f \rangle \rightarrow f = 0$. This condition is checked at the same program points as the staleness condition.

```
1 int output = getValue();
2 while(...) {
3   if(Config.get("op") == "+") {
4     output += 2
5   } else {
6     output *= 2;
7   }
8 }
```

Figure 6 A simplified example of DCU error involving control-flow. If the `"op"` option is changed during the execution of the loop, the `output` variable will be processed inconsistently.

Control-Flow

We have so far discussed the consistency condition in terms of *values* produced and read by a program. This misses inconsistent behavior introduced by control-flow. A simplified example, based on a real bug found in our evaluation set, is shown in Figure 6. In this example, if the `"op"` property is updated before the loop has terminated, the `output` variable can reflect two different versions of the configuration. Lillack et al. [20] have noted that configuration values are often used in branching decisions so detecting errors involving control-flow is especially important.

To find errors like the one in Figure 6, we verify that the entire execution of a method observes a consistent view of the configuration. To check this property, upon entrance to a method we allocate a special sentinel object Σ. This object has a configuration history, which is empty at method entry. Unlike regular configuration histories, which record which configuration options and versions were used to construct a *single* value, $\Sigma^{\mathcal{H}}$ records the options and versions used by the *entire* method. Whenever a value v is read, $\Sigma^{\mathcal{H}}$ is updated such that $\Sigma^{\mathcal{H}'} = \Sigma^{\mathcal{H}} \cup v^{\mathcal{H}}$. $\Sigma^{\mathcal{H}'}$ is immediately checked for consistency using the rule described above. At method exit, Σ and its configuration history are discarded; each invocation begins with a fresh history. Method histories are not inherited by callers or callees.

To see how this approach catches errors involving control-flow, consider again the example in Figure 6. Assume at the entrance to the loop $\Sigma^{\mathcal{H}}$ is initially empty. On the first iteration of the loop, the `Config.get("op")` call on line 3 returns a configuration value with the configuration history $\{\texttt{"op"} \to \langle 1, 0\rangle\}$. This is merged with Σ's empty configuration history, giving $\Sigma^{\mathcal{H}} = \{\texttt{"op"} \to \langle 1, 0\rangle\}$. At the end of the first iteration, `"op"` is updated. On the second iteration, the value returned by the `get()` call on line 3 is a newer version of the `"op"` option, which is tagged with the history $\{\texttt{"op"} \to \langle 2, 0\rangle\}$. This history is merged with $\Sigma^{\mathcal{H}}$. The result of this merge is $\Sigma^{\mathcal{H}} = \{\texttt{"op"} \to \langle 1, 1\rangle\}$. Notice that the consistency flag for `"op"` is now set: this signals that the execution has now observed two conflicting versions of the `"op"` option. Staccato immediately reports this as an error.

The assumption that a method execution is the single unit for consistency is a heuristic, which may admit false positives and false negatives. A more precise approach would require precise interprocedural tracking of control-flow dependencies. We experimented with a version of Staccato that integrated an existing off-the-shelf implementation of precise control-flow tracking found in recent versions of Phosphor [4]. Although this approach was promising on small test examples, it suffered scalability issues when applied to the programs in our evaluation set. For example, after applying Staccato with control-flow tracking to a web application, the program failed to return a response to the client after 5 minutes. This overhead is due to the large amount of state maintenance required for precisely tracking control-flow dependencies. In contrast, our approach is relatively cheap to compute and can

find inconsistencies introduced through control-flow. In our experiments, this approach was not the source of any false positives.

4 Automated Bug Avoidance and Repair

Staccato also uses the techniques developed for checking to (semi-)automatically avoid DCU errors. Staccato supports two forms of program repair and automatic bug avoidance. The first mechanism (Section 4.1) automatically repairs buggy program *schedules* that would expose insufficient synchronization between configuration read and write operations. Staccato detects uses of configuration-derived values and automatically delays any configuration updates that would cause those values to become stale mid-use. Staccato also has a limited form of *value* repair (Section 4.2). The programmer may provide a function called by Staccato that repairs stale values when they are discovered. Both mechanisms target reducing violations of the staleness condition. In our experience, this condition is harder to get "right", and therefore benefits the most from automated assistance.

4.1 Coordinating Configuration Reads and Writes

In a multithreaded environment, it is possible for configuration updates to interfere with an ongoing use of a configuration-derived value. For example, a method may read some initially up-to-date variable x and then later re-read x. If an option in x's configuration history is updated between the two reads, Staccato would identify the second read as an error. It is assumed that the program has failed to appropriately react to the configuration update. However, the *use* of x—and by extension the options in x's configuration history—intuitively spans the two read operations. The error occurs because the *update* operation failed to wait for the outstanding use of x (and its corresponding configuration options) to finish. In general, an update of a configuration option being used in another thread without proper synchronization can lead to a DCU error.

Staccato can automatically *prevent* these errors by using a set of per-option read/write locks, called "option locks". A read acquisition of an option lock indicates that a thread is currently using that option (or some value derived from it). Updates to a configuration option must first acquire the option's lock in write mode. If the option being updated is in use, the write acquisition will stall until all outstanding uses of the option finish and all read holds are released. The option write lock is immediately released after update is effected. The programmer is not responsible for acquiring the write locks during configuration update: all updates are delegated to our implementation's runtime which handles locking.

Read-acquisition of option locks is automatically performed by Staccato during a staleness check. Recall that staleness checks occur after every value read. After a value is read, but before the staleness check is performed, Staccato read-locks the option locks for every option in the read value's configuration history. These locks are held in read mode after control returns to the host program. The scope of a value's use is approximated as the method containing the initial read of that value. Thus, the read-acquisitions on the option locks acquired during a read are held until the containing method completes. Computing the scope precisely is not possible, so we use end-of-method as heuristic. However, using our heuristic was sufficient to prevent several errors that Staccato would have otherwise occurred during our evaluation.

```
class RequestManager implements StaccatoFieldRepair {
  String targetIp, apiKey;
  String doRequest() {
     Request req = new ApiRequest(targetIp);
     req.send(apiKey);
     return req.response();
  }
  Object _StaccatoRepairField(String fieldName,
    Object oldValue, Exception staleException) {
    if(fieldName.equals("apiKey")) {
      return apiKey = Config.get("api-key");
    } else if(fieldName.equals("targetIp")) {
      return targetIp = Config.get("target-ip");
    } else { throw staleException; }
  }
}
```

Figure 7 An updated version of the code in Figure 1 which uses STACCATO repair callbacks. The `Reloadable` interface has been replaced with the `StaccatoFieldRepair`. The new method, `_StaccatoRepairField` is called when STACCATO detects that one of the fields is stale. The updated value returned from the method automatically replaces stale field.

4.2 Value Repair

STACCATO's value repair is an extension to the staleness checking mechanism. Unlike schedule repair, value repair is not fully automatic: we require the programmer to provide a repair function that updates a stale value to reflect the latest version of the configuration. Although STACCATO can automatically *detect* a stale value, automatically *updating* it to the correct value is beyond pure automation because how to respond in the presence of stale data is fundamentally application-specific.

STACCATO's value repair operates exclusively on object fields. STACCATO already intercepts all field reads for checking. When the STACCATO runtime detects that a value read from a field is stale, it checks if the object hosting the field implements an interface that provides the STACCATO update callback. If the callback exists, STACCATO calls it with the name and current value of the stale field. The callback may rebuild the stale field in an application defined way. The updated field value is then returned from the hook and transparently replaces the old, stale value as the result of the original field read. If the callback is unable to repair a field, the original error is reported as usual; failure to repair a field is not itself an error. Figure 7 shows a simplified application of this update mechanism to the example given in Figure 1 in Section 2.

Repairing (or updating) a stale configuration derived value will likely use the same options used to construct the old value. As a convenience, STACCATO calls the update callbacks while holding the option locks described in Section 4.1 in read mode. Thus, although update callbacks may execute in a multithreaded context, the callback effectively sees a consistent, immutable view of the necessary configuration options during the rebuild operation. As a further convenience, STACCATO ensures that no threads concurrently execute the update callback on the same object.

5 Implementation

We have implemented our technique in a prototype tool named STACCATO. STACCATO is an offline bytecode instrumentation tool for Java programs. STACCATO modifies a program's bytecode to support tracking configuration histories and also inserts code to perform check-

ing and propagation. The configuration tracking of STACCATO is built on top of a modified version of the Phosphor tool by Bell and Kaiser [4]. STACCATO also includes a runtime library that implements the operations described in Section 3. STACCATO does not automatically integrate with the program's configuration abstraction, this must be performed by the programmer when initially integrating STACCATO into the software (see Section 5.3).

5.1 Basic Operation

Applying STACCATO is a two-step process. First, our modified version of Phosphor instruments the source program to add information tracking for primitive types. In a second pass, STACCATO uses ASM [19] and Javassist [7] to add code that calls into the STACCATO runtime to perform the check and propagation operations described in Section 3. For each method covered by a correctness condition, STACCATO instruments all array, variable, and field reads, as well as method calls to check read values against the correctness condition selected for the method. STACCATO also inserts code at the end of methods annotated with `@StaccatoPropagate` to perform history propagation. Phosphor already adds unconditional propagation from source values to outputs for primitive instructions such as integer addition; we replaced the merge operation used by Phosphor for these instructions with a call into our runtime library.

STACCATO does not require a modified JVM. To ensure information is soundly tracked through calls to the Java Class Library, a program must use a version of the JCL that has been instrumented by Phosphor. To ease integration, we also added propagation to certain "primitive" operations in the JCL, such as string concatenation or string-to-integer parsing. To use STACCATO, a program must be launched with a special JVM flag that adds our instrumented versions of the system classes to the system classpath.

5.2 Tracking Configurations

To reduce memory overhead, STACCATO does *not* unconditionally instrument every type to carry configuration history. STACCATO uses the programmer's `@StaccatoPropagate` annotations to avoid instrumenting types that will provably never carry configuration information. For a type that *may* carry configuration information, STACCATO adds a special field to the class definition. STACCATO also modifies the class definition to implement an interface that marks the class as carrying configuration information. This interface exposes two methods that provide the functionality to get and set the hidden field. STACCATO synthesizes these methods and inserts them into the class definition. For primitive values (which do not have fields), we reuse the shadow taint variables added by Phosphor.

We chose the interface approach over using reflection for two reasons. First, determining if a type carries configuration information reduces to a relatively inexpensive `instanceof` check. The second reason is to integrate with other dynamic bytecode rewriting tools. There are several libraries that perform instrumentation at runtime to introduce features like database connection pooling.[4] These tools introduce wrapper classes that encapsulate the original configuration carrying values. This hiding makes access to a hidden tag field via reflection impossible. However, we found that these tools often make an effort to preserve the interfaces of types being instrumented. This preservation is done at the type level (a wrapper class for some type T also implements the same interfaces as the original type

[4] e.g., `http://proxool.sourceforge.net/`

T) and in terms of the behavior of the generated class (by delegating method calls to the wrapped value). Using interfaces allowed STACCATO to seamlessly integrate with these dynamic tools. There is no guarantee of interoperability but we found that using interfaces worked with all the dynamic tools we encountered.

5.3 Host Program Integration

STACCATO does not manage the configuration of the software being analyzed: it is the responsibility of the programmer to integrate the STACCATO runtime with the software's configuration abstraction. STACCATO assumes that the software uses a key-value data structure for configuration information. The key-value abstraction is widely used in practice for storing configurations, including the Java Properties API, the Windows Registry, and several real-world software projects [33, 16]. All of the software configurations in our evaluation used this abstraction. Rabkin et al. have shown that complex, hierarchical configurations can be adapted to the key-value model [33]. We also assume that all configuration values are strings. In principle, our approach and implementation could be extended to support configuration values of arbitrary types. In practice, all programs we encountered used strings for storage and performed parsing/serialization of these strings where necessary.

Integrating a program's configuration with STACCATO requires only minimal source changes. The programmer must delegate all set, get, and delete operations on the configuration key-value store to the STACCATO runtime. For example, suppose a program's configuration abstraction is stored in a `HashMap` variable named `conf`. Delegating get operations to STACCATO involves changing calls of the form `conf.get(key)` to `Staccato.get(conf, key)`. Similar changes are made for delete and set operations. The delegation ensures that STACCATO increments option epochs on configuration update and correctly tags configuration values read by the program. The API exposed by the STACCATO runtime is very general; we were able to incorporate all configuration abstractions we found in our evaluation.

5.4 Propagation Coordination

Multithreaded execution introduces the possibility for two or more propagation operations to occur concurrently. If two or more propagation operations involve the same object concurrently, the STACCATO runtime uses locking to impose an *arbitrary* order on configuration history propagation. The history merge operation (see Sections 3.2 and 3.4) is commutative: two or more propagation operations with the same target object can occur in any order without changing the final configuration history associated with the object.

Adding locking to arbitrary user programs risks introducing deadlocks, so we modeled the locking protocol of STACCATO in the Alloy model finder [15]. Our verification of STACCATO's locking was bounded, but we used significantly large parameters to achieve high confidence in our results. Using this model, we verified that the locks introduced by STACCATO do not deadlock. STACCATO's locks can interact with a program's existing locks to produce deadlocks. This is because a thread of execution may hold several option locks in read mode, and perform arbitrary lock acquisitions. In practice, this was never a problem in our evaluation.

5.5 Polymorphism and Subtyping

STACCATO does not instrument every read within a method, only those it cannot prove will not carry configuration information. During instrumentation, STACCATO may encounter

a value of type T that does not itself track configuration information, but there is some subtype of T that does. In this scenario, STACCATO cannot statically determine if checking should be performed. STACCATO handles this ambiguity by inserting *conditional* checks. If the runtime type of an object does not carry configuration information, a conditional check is a no-op, otherwise a regular check is performed.

Java generics pose a similar problem. Java implements polymorphism using type erasure [5]: `Object` is used to represent type variables at the bytecode level. As every type is a subtype of `Object`, the use of type variable in a target program will lead to ambiguity for STACCATO's instrumentation. However, this situation is simply a degenerate case of the subtyping ambiguity described above. STACCATO therefore resolves all ambiguity caused by generics by exclusively using conditional checks.

5.6 Limitations

STACCATO inserts the code that performs the propagation operation at method return after the method body has completed. The propagation target's object's configuration history will therefore be updated *after* the object state has been changed. As a result, the propagation is not necessarily atomic with the body of the method.[5] This can result in the configuration history for an object briefly not reflecting the current state of the object. Unfortunately, it is impossible for STACCATO to automatically incorporate the propagation operation into a synchronization scheme for a method. This limitation did not prevent us from effectively finding bugs and did not admit any false positives.

The association of configuration histories with program values requires that two logical values with distinct configuration histories must have unique identities. This restriction means that STACCATO interacts poorly with memoization or singleton objects, as two or more logically separate configuration histories may be conflated in the same program value. This was primarily a problem for literal strings and enumeration variants. All occurrences of the same string literal and enumeration variant within a single JVM instance have the same object identity. For literal strings, boolean options were particularly problematic: boolean configuration values were almost always stored using the literals `"true"` and `"false"`. STACCATO conservatively performs a deep-copy when it detects it would otherwise propagate configuration history to a string literal or enumeration.

However, copying enumerations will break a program that relies on the referential equality between two enumeration variants with the same name. To work around this, unboxing operations are added to equality tests that involve enumerations. Unboxing restores a copied enumeration back to the original singleton object for purposes of comparison. This unboxing imposes a non-trivial overhead, and the programmer must opt-in to this mechanism.

Finally, incorrect annotations by users can produce incorrect analysis results. Over-annotating can lead to false positives. However, these false positives reveal where the programmer's assumptions (expressed in his/her annotations) about how a program handles DCU are incorrect. Under-annotation results in bugs being missed, but existing analyses that track the flow of configuration values, e.g. Lotrack [20], and ConfAnalyzer [32], can help guide the user to the correct annotations. This limitation is inherent and would require a user-study to measure the extent of the problem in practice.

[5] If the method is marked as `synchronized`, the propagation is protected by the object monitor that protects the entire method body.

6 Evaluation

Having defined our approach to finding bugs in dynamic software configuration, we present our evaluation of STACCATO's effectiveness. In evaluating STACCATO, we were interested in the following 4 questions:

1. Does software with dynamic configurability have violations of our correctness conditions?
2. How effective is STACCATO at finding these errors?
3. For some real applications, what is the annotation burden of using STACCATO?
4. What is the performance impact of using STACCATO?

We focused our evaluation on open-source applications with high-levels of run-time configurability. We chose three projects of substantial size but approachable complexity: Openfire (version 3.9.3),[6] a full featured chat server that implements the XMPP IM protocol, JForum (version 2.1.8),[7] a widely deployed forum software, and Subsonic (version 5.2.1),[8] a music streaming server. Each project has many configuration options and extensive concurrency.

We did not perform an exhaustive evaluation on Solr as the code-base was close to 500,000 SLOC: considerably larger than our next largest evaluation target (Openfire). No technical limitation prevents using STACCATO on Solr: the instrumentation process and dynamic analysis scale independently of code-base size. However, using STACCATO requires understanding a code-base and 500,000 lines felt beyond what we could reliably do ourselves. We did perform an informal evaluation on Solr where we tried to detect the bug mentioned in the introduction using STACCATO. We needed only a handful of propagation annotations and one check annotation to re-find the bug. We believe, but haven't substantiated, that a team familiar with Solr could use STACCATO without undue burden.

Experimental setup

We manually annotated each application after first becoming familiar with the code-base and how the software uses configuration values. We also integrated each software's configuration abstraction with the STACCATO runtime.

We were unable to find extensive functional tests for any of the projects. We developed our own functional tests for each of the three software projects. Due to the differences in the software under evaluation, we had to use different evaluation techniques depending on the software. For Openfire, we used Tsung[9] version 1.4.2. For JForum and Subsonic we used Apache JMeter[10] version 2.12. We developed test plans in these tools that exercised core functionality of each software (these tests are included in the STACCATO distribution). Each functional test client executes in a loop. On each iteration of the loop, a test clients sleeps for a short, randomly selected period of time and then performs a randomly selected test action. These test actions were prepared by us and were designed to use one piece of the core functionality of the software under test. For example, one of the test actions for JForum involves sending a private message from one user to another.

To induce configuration errors, we also developed a *havoc mechanism*. This havoc mechanism consists of several test clients that execute alongside the functional test clients. Each

[6] `http://www.igniterealtime.org/projects/openfire/index.jsp`
[7] `http://jforum.net/`
[8] `http://www.subsonic.org/pages/index.jsp`
[9] `http://tsung.erlang-projects.org/`
[10] `http://jmeter.apache.org/`

havoc client executes in an infinite loop. During each iteration of the loop, the havoc client sleeps for a short, randomly selected period of time and then performs a mutation to the software's configuration. This mutation was done by simulating an HTTP request to the administrative webpages of the software under test. These administrative webpages also validated our havoc updates (e.g., by rejecting attempts to select negative port numbers). The havoc clients were also restricted to choosing mutations that we had manually prepared in consultation with the program code and documentation. Any errors reported by STACCATO during testing were logged for collection. We report our findings from these experiments in Section 6.3.

We evaluated the performance impact of STACCATO along two dimensions, slowdown and memory overhead. To calculate the slowdown, we ran each software's test suite on the instrumented and uninstrumented versions 5 times each and measured response times. Before collecting any data we ran a shorter version of each project's test suite to control for the effects of JIT compilation and application-specific startup actions. We disabled the havoc mechanism during performance testing, as the high rate of configuration updates (several times per second) is not realistic for measuring performance. To measure the memory overhead of STACCATO, we ran the same test suites (again with a brief warmup period) and used Java's JMX technology to monitor the program's memory usage. We sampled the JVM's reported memory usage at one second intervals. Before taking each measurement we triggered a garbage collection.

All experiments were run on a Dell Latitude E-5440 with a 4 core Intel Core i5 processor at 2.00 GHz and with 16GB of RAM. We used version 1.7.0-91 of the OpenJDK JVM.

6.1 Summary

The annotation burden of using STACCATO is extremely low: the ratio of annotation to SLOC is below 1%. Even including changes to integrate with STACCATO and update handlers, we found that the effort needed to use STACCATO in a project is minimal. Further, STACCATO was able to accommodate most configuration options of interest and usage patterns in the projects that we evaluated.

We found DCU errors in all of our evaluation targets, despite widely different approaches to configuration discipline and multithreading. Many of the DCU errors that we found were violations of the staleness condition caused by two concurrent configuration updates. Two of the projects we evaluated with STACCATO also had DCU errors that could occur in a single-threaded context, indicating that DCU errors are not just the result of insufficient testing of multithreaded software. We encountered only one false positive during our analysis.

STACCATO imposed a moderate performance penalty in our tests; we measured a maximum overall slowdown of 5.30x and a memory overhead of at most 144.40%. This is overhead low enough to use STACCATO as a bug-finding tool in pre-deployment. Our experience suggests combining STACCATO with an automated havoc test like those used in our evaluation can be an effective technique for finding DCU errors. STACCATO's overhead makes it unlikely that the repair mechanisms described in Section 4 can be used in production. However, a developer can use STACCATO's repair mechanisms to rapidly develop DCU implementations or fixes. From an initial implementation using STACCATO, a programmer can develop a more efficient manual solution.

Table 1 Counts of lines changed to integrate with STACCATO. **Annot.** counts explicit check and propagation annotations. **Flow** are changes to calling conventions, the introduction of helper methods, or the use of wrapper classes for integration with STACCATO. **Repair CB** are repair callback code. **Bug-Fixes** are fixes for concurrency bugs in the project. These consist primarily of field accesses that were not well ordered according to the Java Memory Model [26]. **Conf-Abs.** are changes to integrate the project's configuration abstraction with the STACCATO runtime. **SLOC** counts the total source lines of code in the original, uninstrumented project.

Project	Annot.	Flow	Repair CB	Bug-Fixes	Conf-Abs.	Total	SLOC
Openfire	100	184	451	78	212	1,025	85,416
JForum	11	44	13	2	29	99	29,568
Subsonic	13	52	0	0	118	183	29,592

6.2 Integration Effort

As a proxy for programmer effort, we measured the number of line changes necessary to integrate STACCATO with each of our evaluation targets. We count explicit propagation and checking annotations as well as lines changed to integrate the software's configuration abstraction with STACCATO's runtime, changes to calling conventions to ease application of annotations, and repair callbacks. The breakdown of lines changed is shown in Table 1. The ratio of annotations to total lines for each project is 0.12%, 0.04%, and 0.04% for Openfire, JForum, and Subsonic respectively. Including all source lines changed gives: 1.20%, 0.33%, and 0.62% for Openfire, JForum, and Subsonic respectively. The relatively high percentage for Openfire can be attributed to large number of lines added for update callbacks (column **Repair CB** in the table).

Qualitatively, most of our effort was spent understanding each code-base. A team familiar with an application would be spared this effort. Adapting a program once we understood how its dynamic configuration worked was largely formulaic. Many bugs were found with no extra annotations as many methods are checked by default against the consistency condition. Finding staleness violations required some manual annotation. However, recall that annotations to control checking by STACCATO can be applied to an entire class. Thus, when we added annotations for finding staleness violations, no pre-existing knowledge of exact bug location was required, only an intuition that a class encapsulated some persistent, configuration-derived state. Finally, we found most candidates for propagation annotations simply by inspecting configuration access sites and determining what methods were called with the newly returned configuration values.

Checking Coverage

We found that the configuration model and primitives exposed by STACCATO were sufficient to achieve good coverage of options checked in each evaluation target. Our functional tests exercised between 25%–86% of options in our evaluation programs. STACCATO was able to track and check all options exercised by our tests.

We measured coverage by modifying the STACCATO runtime to record the options checked by STACCATO and the correctness condition being checked. We also recorded which options were read or updated during test execution. Options not involved in at least one update operation are not counted for coverage purposes. Uses of these options were trivially validated by STACCATO but are uninteresting for evaluation purposes. Options that were updated but never re-read *are* included; STACCATO still validated that old copies of the option were used consistently and did not cause violations of the consistency condition.

Table 2 Classification of options available in each application and coverage of our functional test suite. **Checked** counts options checked during our tests. **Update** are options updated using Staccato's repair mechanism. The next 3 columns count options not included in our tests. **Imm.** are immutable and therefore uninteresting for our evaluation. **Int.** are internal, undocumented options for which we lacked sufficient domain knowledge of the software. **Other** are options that were untestable because of missing proprietary technology or update mechanisms with unfixable concurrency bugs. **Untested** are all remaining options not exercised by our functional tests. **Cov** measures the percentage of options tested in our evaluation.

Project	Checked	Update	Imm.	Int.	Other	Untested	Total	Cov.
Openfire	24	21	68	73	4	61	251	25.14%
JForum	37	1	3	0	0	6	47	86.36%
Subsonic	33	0	0	0	0	15	48	68.75%

Table 3 Counts of errors found by Staccato. **Stale Read** are violations of the staleness condition. **Incons. Read** are violations of the consistency condition. **Incons. CF** are violations of the consistency condition for control-flow.

Project	Stale Read	Incons. Read	Incons. CF
Openfire	11	0	0
JForum	0	5	0
Subsonic	1	2	1

In all tests, every configuration option mutated by the havoc mechanism was checked at least once. This result indicates that our approach accurately models the configuration updates of the evaluation programs. Across all projects, most options were checked against the consistency condition rather than the staleness condition. However, all programs used checks for staleness at least once. Openfire and Subsonic both used the condition to find bugs and JForum used staleness checking in combination with repair to add support for configuration updates.

Our tests exercised only a representative set of options for each program in our evaluation. There is no restriction other than annotation and test development effort preventing 100% coverage of options. We felt that achieving full coverage in our tests would provide low marginal benefit for the amount of effort required. For example, we did not test the configuration options that control Openfire's operation in multi-server clusters. An overview of which options we checked during our evaluation is presented in Table 2. We calculate the coverage of our tests as:

$$\frac{Checked + Update}{Checked + Update + Untested + Internal}$$

Coverage for our evaluation set is 25.14% 86.36% and 68.75% for Openfire, JForum, and Subsonic respectively. Openfire's relatively low coverage is due to the high number of options available in that program. Some of these options (counted in column Untested) controlled functionality that was difficult to test automatically or that required complex test environments (e.g., mutli-server cluster options). We also excluded many undocumented options available in Openfire that cannot be set via normal means (counted in column Internal).

```
public void setLogDir(String directory) {
  Config.set("audit.log.dir", directory);
  this.auditDir = directory;
}
```

Figure 8 Simplified atomicity bug during configuration update. If two threads concurrently execute `setLogDir`, the value in the `auditDir` field may not be the most recent version of `"audit.log.dir"` option. This violates the object's implicit invariant. STACCATO detects this error on subsequent reads of the `auditDir` field.

```
static Map<String, ContactList> cache = ...;
ContactList getContactList(String user) {
  if(cache.containsKey(user)) {
    return cache.get(user);
  } else {
    ContactStorage store = getContactStorage();
    cache.put(user, new ContactList(user, store));
    return list;
  }
}
ContactStorage getContactStorage() {
  String backend = Config.get("contact.classname");
  /* reflectively instantiate the backend specified by
  "contact.classname" */
}
```

Figure 9 Simplified example of contact list bug in Openfire. A stale version of the backend specified by `"contact.classname"` will persist in ContactStorage objects cached in `cache`.

6.3 Effectiveness

Table 3 summarizes the errors that we found in our three evaluation targets. In every evaluation target, STACCATO found multiple dynamic configuration update errors. In total, STACCATO discovered 20 errors across the three evaluation programs. The most common bugs are staleness violations that occur when two concurrent configuration updates were performed. However, two of the three projects also contain errors that do not depend on concurrency to manifest. We now discuss some example bugs found in each evaluation target.

6.3.1 Openfire

All of the bugs found in Openfire were violations of the staleness condition. The most common error in Openfire was reads of out-of-date configuration data caused by non-atomic configuration updates. Figure 8 shows a simplified example of the problematic configuration update idiom behind these errors.

We also found a DCU error in Openfire that does not require a specific thread schedule. The server administrator may configure, at runtime, the implementation used to store a user's contact list. Each user's contact list object holds a reference to the storage backend specified by the administrator. STACCATO found that contact lists loaded before a change to the storage backend would continue to use the old backend object. This could cause changes made to a user's contact list to be lost. The underlying defect was caused by the contact list objects being cached in a static field. A sketch of this scenario is shown in Figure 9.

Using the repair callback mechanism described in Section 4, we were able to introduce dynamic updates for 21 options. We also added repair for 2 of the errors that we found in

our evaluation. The locking guarantees provided by STACCATO allowed us to write update callbacks that focused only on implementing the actual configuration update and did not require us to add any extra locking. STACCATO also ensured that we did not introduce any new dynamic configuration update errors when adding this new DCU functionality. We validated that our update code was correct by extending the havoc mechanism to include changes to the configuration options for which we wrote update callbacks; no errors were reported by STACCATO in any of the options we tested in this way.

6.3.2 JForum

JForum maintains less configurable global state than Openfire, so STACCATO did not find any violations of the staleness condition. However, STACCATO found 5 violations of the consistency condition.

Most of these errors were the result of two or more reads of the same configuration option without any synchronization. For instance, when generating a response, JForum reads an option twice to set two separate response parameters. The two reads are not protected by any synchronization, and STACCATO detected that a concurrent update of the option would yield an inconsistent response. The single false positive found during our evaluation occurred under similar circumstances. JForum reads a single option twice (again, without synchronization), except that both reads populate the *same* response parameter. STACCATO was unable to detect that the second write overwrote the first, concealing any inconsistency.

One of the consistency errors STACCATO discovered caused JForum to mislabel the encoding used in an HTTP response. Finding this error involved tracking object dependencies at a level not possible with just primitive value dependencies. The bug is caused by the configuration option that controls response encoding being read twice during request handling: once in the method that generates the HTML response, and in another method that sets the response headers. Using to `@StaccatoPropagate` annotations, STACCATO was able to capture the dependencies between the encoding option, the generated output, and the response headers.

STACCATO also found a consistency error that does not rely on a specific thread schedule to manifest. JForum pre-computes the URLs of emojis available on the forum and stores the results in a cache. The emoji URLs are built in part using the URL of the forum, which is configurable. However, after the forum URL is updated, the emoji URLs in the cache are not updated. The forum URL is also read on each request to render the header and footer content of each page. STACCATO detected that after the forum URL is updated, responses that include emojis contain two inconsistent versions of the forum URL.

The handful of options that control persistent state all require a restart to take effect, indicating a conservative approach on the part of the JForum developers towards configuration updates. We were able to use STACCATO's repair functionality to transparently and safely introduce online updates for one of these options.

6.3.3 Subsonic

Compared to JForum and Openfire, Subsonic has little configurable global state. The one exception was in the LDAP integration component. Subsonic caches its connection to an LDAP server in a static field and rebuilds the connection when it detects that the LDAP configuration has changed. A simplified form of this update algorithm is shown in Figure 10. However, although the `getConnection` method is synchronized, the method does not synchronize with updates to the software configuration. As a result, `lastChecked` may hold the most

```
1 static LdapConnection ldapConnection = ...;
2 static Time lastChecked = ...;
3 synchronized LdapConnection getConnection() {
4   if(lastChecked < Config.lastUpdateTime()) {
5     ldapConnection = Ldap.connect(Config.get("ldap-config"));
6     lastChecked = Config.lastUpdateTime();
7   }
8   return ldapConnection;
9 }
```

Figure 10 A sketch of the Subsonic LDAP connection staleness bug. In this example `Config.lastUpdateTime()` returns the time of the most recent update to the configuration. Although the update code uses `synchronized`, the update still contains an error. If an update to the `"ldap-config"` option occurs between lines 5 and 6, the program incorrectly concludes that the `ldapConnection` field is up-to-date.

recent update time for the configuration but the `ldapConnection` could contain a connection built with an old version of the `"ldap-config"` option. Notice that this idiom is very similar to the value repair scheme described in Section 4.2, although the option locks (Section 4.1) prevent a smiliar error from occurring within our implementation.

Subsonic also contains a control-flow consistency error. A method that controls the music play queue iterates over a list of playlist items and on each iteration checks an option that determines if Subsonic is running behind a firewall. If this option is set, extra processing is performed on the playlist item. Staccato flagged that if the option was changed during an iteration of the loop, Subsonic would inconsistently process the entire list as a result.

One final example bug found by Staccato was caused by one logical option (the server locale) being stored across three different configuration options. The reads of the three options where not synchronized with the update mechanism of the options. As a result, if a read of the three locale options occurred concurrently with an update, Staccato detected that the user would receive an invalid locale (e.g, the US version of Japanese).

6.4 Performance Impact

Finally, we measured the performance impact of Staccato on our evaluation targets. We measured two metrics: memory overhead and overall slowdown. Across all evaluation programs, the largest overall slowdown (averaging over test actions) was 5.30x and memory overhead was below 2x. A graph of the slowdown results can be found in Figure 11.

We calculated overall slowdown as the geometric mean of the ratio between average response times with and without Staccato, after excluding the largest 5% of response times as outliers. Tsung (the technology used to test Openfire) did not provide individual response times, so we did not filter outliers when calculating Openfire's slowdown. The number and duration of the outliers was broadly similar with and without Staccato, with high variance across runs.

The overall slowdown for the projects was 1.26x, 5.30x, and 2.11x for Openfire, JForum, and Subsonic respectively. This is generally competitive with other dynamic analyses and is consistent with the performance reported for the Phosphor tool [4] on top of which Staccato is built.[11] The majority of extra time is spent combining configuration histories for

[11] The slowdown reported in the Phosphor paper is for a version that supported only integer valued tags. Staccato is built using a more recent version that supports arbitrary tag types, which is necessarily slower.

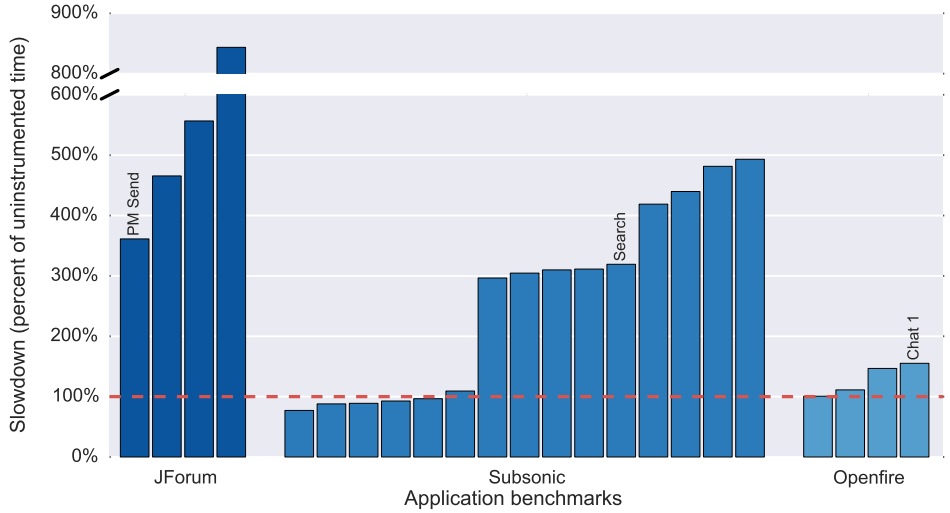


Figure 11 The average slowdowns for the evaluation applications. Each individual bar represents the average slowdown of a single test action in the test suites we created for the project. For example, the "PM Send" bar measures the average slowdown experienced when sending a private message to another user on the forum. Similarly, the "Search" and "Chat 1" bars respectively measure the average slowdown for searching Subsonic's music catalog and sending a sequence of chat messages on the Openfire server. The overall slowdown for each application is computed by taking the geometric mean of these average slowdowns.

primitive types. History merging is performed via a relatively expensive method call that is inserted after every arithmetic, floating point, and boolean operation. In the common case, the configuration histories being merged are empty. Although we optimized our runtime heavily for this common case, STACCATO runs into the inherent overhead of method calls. STACCATO could be combined with a whole-program static analysis to prune merge operations when the configuration histories involved are provably empty.

The memory overhead of STACCATO was: 110.83%, 144.40%, and 114.43% for Openfire, JForum, and Subsonic respectively. For each application, we calculated the memory overhead by averaging the observed heap sizes during the instrumented test runs, and similarly for the uninstrumented test runs. We took the ratio of these two averages to be the total memory overhead. Most of this memory overhead is added by Phosphor for shadowing primitive variables as noted in the Phosphor paper. Our memory overhead is generally lower than that reported for Phosphor as we do not add tag fields unconditionally to every object as explained in Section 5.2. We expect that memory usage could be improved with a static analysis to remove provably empty shadow state.

7 Related Work

Configuration management is an active area of research [42, 39, 40, 1, 2, 16, 43, 9, 34, 27, 20, 33, 41]. To our knowledge, no existing research has examined the problem of DCU errors. Some existing work has used approaches similar to ours to study different problems. ConfAid [3] uses a dynamic information-flow analysis similar to ours to diagnose software misconfigurations. It associates each program value with a *configuration set*: this set tracks which configuration options potentially influenced the construction of the value. This is very similar to STACCATO's configuration histories. However, ConfAid reasons only about static configurations, and does not track versions like STACCATO. Rabkin et al. [32] also

tracked configuration values for the purposes of diagnosing configuration errors. However, their analysis does not reason about dynamic configuration updates, and uses a conservative static analysis.

Staccato's analysis builds on extensive work on dynamic information-flow analysis [11, 12, 4]. Our approach precisely tracks data dependence, but handles handles dependencies from indirect flow using a heuristic. Dynamic analyses that precisely track dependencies from control-flow do exist (e.g., [18, 8]) but suffer scalability that limits their application to large programs such as the ones we use in our evaluation. However, given a scalable framework for control-flow tracking, Staccato could be extended to support control-flow.

Several of the DCU errors we found were the result of atomicity violations, such as the JForum response parameter bug, or the example in Figure 8. These errors are similar to linearizability errors. There is research into automatically checking linearizability [35, 6, 36] and repairing operations that are not linearizable [23]. However, existing linearizability research focuses on linearizability of ADT operations and cannot handle the arbitrary operations performed on configuration options. Current work on dynamic atomicity checkers [10, 38] offer another possible solution to detect consistency violations involving configurations. However, linearizability and atomicity checkers would not find DCU errors that involve our staleness condition.

Staleness violations result from a DCU invalidating one thread's assumptions about the (global) configuration state. This is similar to check-then-act errors [22, 21, 29] that result from a concurrent update invalidating assumptions established by a check operation. In contrast to check-then-act errors, Staccato's consistency condition allows concurrent updates to the global configuration, provided that a thread never observes inconsistencies.

Finally, there has been considerable research in the field of dynamic software updates (DSU) [24, 13, 30, 25, 14, 31, 28]. DSU attempts to introduce arbitrary code changes to running software without service interruption. Dynamic configuration updates can be viewed as a specific case of DSU: each configuration update is a controlled change to running software at runtime. However, existing DSU research does not provide checking for bugs in a software's current dynamic configuration mechanism, and offers only a potential alternative for current DCU mechanisms. Unfortunately, even in the state of the art, existing DSU techniques require significant effort on the part of the programmer to integrate with existing applications and impose non-trivial performance overhead. One interesting common point between existing DSU research and the approach in Staccato is how to maintain consistent application state in the presence of an online update. Many existing approaches require the programmer to write update hooks (e.g., [28]): this is similar to the approach taken by Staccato for program repair (Section 4.2).

8 Conclusions and Future Work

This paper presented the first study of errors in dynamic configuration updates. We attacked the problem of diagnosing incorrect updates with a dynamic analysis tool Staccato, which detects stale or inconsistent views of the software configuration. We evaluated Staccato on three open-source projects and found bugs in all three. Staccato imposes moderate performance overhead, and requires only low manual annotation overhead. In future work, we hope to complement Staccato with static analysis to lower the need for test cases to find DCU errors.

Acknowledgments. The authors thank James Bornholt, Doug Woos, Pavel Panchenkha, James Wilcox, and Emina Torlak for helpful discussions and feedback on early drafts of

this paper. We would also like to thank Jonathan Bell for his help with the Phosphor tool and adding support for enumerations. Finally, we thank the anonymous reviewers for their helpful feedback.

References

1 Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.

2 Mona Attariyan and Jason Flinn. Using causality to diagnose configuration bugs. In *USENIX Annual Technical Conference*, 2008.

3 Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.

4 Jonathan Bell and Gail Kaiser. Phosphor: illuminating dynamic data flow in commodity JVMs. In *OOPSLA*, 2014.

5 Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA*.

6 Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: A complete and automatic linearizability checker. In *PLDI*, 2010.

7 Shigeru Chiba. Javassist-a reflection-based programming wizard for Java. In *OOPSLA Workshop on Reflective Programming in C++ and Java*, 1998.

8 James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA*, 2007.

9 Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with iTuned. *Proceedings of the VLDB Endowment*, 2009.

10 Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, 2008.

11 Malay Ganai, Dongyoon Lee, and Aarti Gupta. Dtam: dynamic taint analysis of multi-threaded programs for relevancy. In *FSE*, 2012.

12 Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for Java. In *Computer Security Applications Conference*, 2005.

13 Christopher M Hayden, Karla Saur, Michael Hicks, and Jeffrey S Foster. A study of dynamic software update quiescence for multithreaded programs. In *Workshop on Hot Topics in Software Upgrades*, 2012.

14 Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for C. In *OOPSLA*, 2012.

15 Daniel Jackson. Software abstractions: Logic. *Language, and Analysis. MIT Press*, 2012, 2006.

16 Dongpu Jin, Myra B Cohen, Xiao Qu, and Brian Robinson. PrefFinder: getting the right preference in configurable software systems. In *ASE*, 2014.

17 Dongpu Jin, Xiao Qu, Myra B Cohen, and Brian Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *ICSE*, 2014.

18 Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

19 Eugene Kuleshov. Using the ASM framework to implement common Java bytecode transformation patterns. *Aspect-Oriented Software Development*, 2007.

20 Max Lillack, Christian Kästner, and Eric Bodden. Tracking load-time configuration options. In *ASE*, 2014.

21 Yu Lin. *Automated refactoring for Java concurrency.* PhD thesis, University of Illinois at Urbana-Champaign, 2015.

22 Yu Lin and Danny Dig. Check-then-act misuse of Java concurrent collections. In *ICST*, 2013.

23 Peng Liu, Omer Tripp, and Xiangyu Zhang. Flint: fixing linearizability violations. In *OOPLSA*, 2014.

24 Kristis Makris and Rida A Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *USENIX Annual Technical Conference*, 2009.

25 Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *EuroSys*, 2007.

26 Jeremy Manson, William Pugh, and Sarita V Adve. *The Java memory model*, volume 40. ACM, 2005.

27 Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *ICSE*, 2014.

28 Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *PLDI*, 2006.

29 Mathias Payer and Thomas R. Gross. Protecting applications against TOCTTOU races by user-space caching of file metadata. In *VEE*, 2012.

30 Luís Pina, Luís Veiga, and Michael Hicks. Rubah: DSU for Java on a stock JVM. In *OOPSLA*, 2014.

31 Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, and Gunter Saake. Javadaptor—flexible runtime updates of Java applications. *Software: Practice and Experience*, 2013.

32 Ariel Rabkin and Randy Katz. Precomputing possible configuration error diagnoses. In *ASE*, 2011.

33 Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *ICSE*, 2011.

34 Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, 2010.

35 Ohad Shacham, Nathan Bronson, Alex Aiken, Mooly Sagiv, Martin Vechev, and Eran Yahav. Testing atomicity of composed concurrent operations. In *OOPSLA*, 2011.

36 Ohad Shacham, Eran Yahav, Guy Golan Gueta, Alex Aiken, Nathan Bronson, Mooly Sagiv, and Martin Vechev. Verifying atomicity via data independence. In *ISSTA*, 2014.

37 Ya-Yunn Su, Mona Attariyan, and Jason Flinn. Autobash: Improving configuration management with operating system causality analysis. In *SOSP*, 2007.

38 Liqiang Wang and Scott D Stoller. Runtime analysis of atomicity for multithreaded programs. *Transactions on Software Engineering*, 2006.

39 Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J Wang, Chun Yuan, and Zheng Zhang. Strider: A black-box, state-based approach to change and configuration management and support. *Science of Computer Programming*, 2004.

40 Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. Generating range fixes for software configuration. In *ICSE*, 2012.

41 Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *SOSP*, 2013.

42 Sai Zhang and Michael D Ernst. Automated diagnosis of software configuration errors. In *ICSE*, 2013.

43 Sai Zhang and Michael D Ernst. Which configuration option should I change? In *ICSE*, 2014.

Transforming Programs between APIs with Many-to-Many Mappings*

Chenglong Wang[1], Jiajun Jiang[2], Jun Li[3], Yingfei Xiong†[4], Xiangyu Luo[5], Lu Zhang[6], and Zhenjiang Hu[7]

1 **Key Laboratory of High Confidence Software Technologies, MoE and Software Engineering Institute, Peking University, Beijing, 100871, China**
chenglongwang@pku.edu.cn

2 **Key Laboratory of High Confidence Software Technologies, MoE and Software Engineering Institute, Peking University, Beijing, 100871, China**
jiangjiajun@pku.edu.cn

3 **Key Laboratory of High Confidence Software Technologies, MoE and Software Engineering Institute, Peking University, Beijing, 100871, China**
lij@pku.edu.cn

4 **Key Laboratory of High Confidence Software Technologies, MoE and Software Engineering Institute, Peking University, Beijing, 100871, China**
xiongyf@pku.edu.cn

5 **Key Laboratory of High Confidence Software Technologies, MoE and Software Engineering Institute, Peking University, Beijing, 100871, China**
vani@pku.edu.cn

6 **Key Laboratory of High Confidence Software Technologies, MoE and Software Engineering Institute, Peking University, Beijing, 100871, China**
zhanglucs@pku.edu.cn

7 **National Institute of Informatics, Tokyo 101-8430, Japan and Key Laboratory of High Confidence Software Technologies, MoE and Software Engineering Institute, Peking University, Beijing, 100871, China**
hu@nii.ac.jp

Abstract

Transforming programs between two APIs or different versions of the same API is a common software engineering task. However, existing languages support for such transformation cannot satisfactorily handle the cases when the relations between elements in the old API and the new API are many-to-many mappings: multiple invocations to the old API are supposed to be replaced by multiple invocations to the new API. Since the multiple invocations of the old APIs may not appear consecutively and the variables in these calls may have different names, writing a tool to correctly cover all such invocation cases is not an easy task.

In this paper we propose a novel *guided-normalization* approach to address this problem. Our core insight is that programs in different forms can be normalized to a semantically equivalent basic form guided by transformation goals, and developers only need to write rules for the basic form to address the transformation. Based on this approach, we design a declarative program transformation language, PATL, for adapting Java programs between different APIs. PATL has simple syntax and basic semantics to handle transformations only considering consecutive statements inside basic blocks, while with guided-normalization, it can be extended to handle complex forms of invocations. Furthermore, PATL ensures that the user-written rules would not accidentally break def-use relations in the program.

* This work is partially supported by the High-Tech Research and Development Program of China under Grant No.2015AA01A202, and the National Natural Science Foundation of China under Grant No.61421091, 61225007, 61432001.

† corresponding author

30th European Conference on Object-Oriented Programming (ECOOP 2016).
Editors: Shriram Krishnamurthi and Benjamin S. Lerner; Article No. 25; pp. 25:1–25:26

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We formalize the semantics of PATL on Middleweight Java and prove the semantics-preserving property of guided-normalization. We also evaluated our approach on three non-trivial case studies: i.e. updating Google Calendar API, switching from JDom to Dom4j, and switching from Swing to SWT. The result is encouraging; it shows that our language allows successful transformations of real world programs with a small number of rules and a small number of manual resolutions.

1998 ACM Subject Classification D.1.2 Automatic Programming.

Keywords and phrases Program transformation, API migration

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.25

1 Introduction

Modern programs depend on library APIs, and when those APIs change, usually many places of the client programs need to be changed. As a result, it is desirable for API vendors to provide program transformation tools that automate the task of transforming client programs. In practice, such transformation tools are developed for two purposes. One is for API upgrade [11, 19]: when an incompatible new version of an API is released, the tool helps upgrade the client programs to work with the new API. For example, a tool named 2to3 script [30] is provided to help users migrate programs from Python 2 to Python 3. Another is for API switching [34], where the tool migrates programs from one platform to another platform. For example, RIM provides a migration tool for transforming Android applications to Blackberry applications in order to attract more developers to the Blackberry platform.

Given the importance of developing program transformation tools, dedicated program transformation languages have been proposed, such as SmPL [27], Stratego [6], TXL [9], Syntax Macros [38], Twinning [24] and SWIN [18]. These languages typically allow the developers to describe a set of rules for program transformation, and a rule usually consists of a pattern that matches a piece of code and an action for generating new code to replace the matched one.

For example, the rule `rSetAlign` in the left of Figure 1 is a simple rule to transform a method invocation in Swing to its counterpart in SWT. The rule is written in PATL (PAtch-like Transformation Language), the transformation language to be introduced in the paper, with a similary patch-style syntax used in SmPL [27]. This rule matches a call to method `JButton.setAlignmentX` and replaces it by a call to method `Button.setAlignment`. The line marked with '-' describes a pattern to match a statement, while the line marked with '+' describes a pattern for generating statements. The declaration of parameter `jb` declares a metavariable used in the patterns and `JButton->Button` indicates that `jb` is of type `JButton` in the old API and of type `Button` in the new API.

Though program transformation languages have greatly reduced the difficulty of writing program transformations, dealing with many-to-many mappings between APIs remains difficult. *Many-to-many mapping* means that a sequence of invocations to the old API is mapped to a sequence of invocations to the new API, and this mapping is minimal: there exists no mapping from a subsequence of invocations to the old API to a subsequence of the new API. As shown by existing studies [34, 4], *many-to-many mappings* are very common in transformation between APIs. Below we present the two key challenges in dealing with many-to-many mappings.

```
//rule rSetAlign
(jb: JButton->Button, align: int->int){
 - jb.setAlignmentX(align);
 + jb.setAlignment(align);
}
```

```
//rule rButton
(jb: JButton->Button,
 parent: JPanel->Composite) {
 - jb = new JButton();
 - parent.add(jb);
 + jb = new Button(parent, SWT.PUSH);
}
```

Figure 1 Two transformation rules used in transformation of programs between Swing to SWT.

```
1 //Case 1:
2 jb = new JButton();
3 if(parent != null) {
4   parent.add(jb);
5 }
```

```
1 //Case 2:
2 jb = new JButton();
3 s = jb.getUIClassID();
4 parent = new JPanel();
5 parent.add(jb);
```

```
1 //Case 3:
2 jb = new JButton();
3 defaultButton = jb;
4 parent.set(defaultButton);
```

Figure 2 Three examples of non-consecutive API invocations.

Challenge 1. The first challenge is that the same sequence of API invocations at runtime can appear in many different forms in the code, and all such possible forms should be considered in transformation.

Let us consider the rule `rButton` in the right of Figure 1. In this case, the class `JButton` is mapped to the class `Button`, and the two calls in the Swing API are merged into one call to the constructor of `Button` in SWT. While this rule captures the basic forms of consecutively invoking the respective methods, many other different forms may produce the sequential invocation to these two statements, such as those in Figure 2.

In this first case, every time the program enters the if-branch, the calls to `jb = new JButton()` and `parent.add(jb)` actually form consecutive calls defined in the pattern, while in the else-branch, such consecutive calls will not be produced. In the second case, though the two invocations are not consecutive, their behavior is equivalent to consecutive invocations and thus should be transformed. (In this case, the third statement `parent = new JPanel()` has no dependency on the first two statements, nor does the second statement depend on the last two statements. So the whole execution is equivalent to an execution of statements 4, 2, 5, 3, where 2 and 5 are consecutive.) In the third case, though the argument `defaultButton` in the third statement is different from `jb` in the first statement, they match the patterns in the rules as these two variables are aliases and they both refer to the same object. None of the three cases are directly captured by rule `rButton`.

Researchers have noticed this problem and have proposed different mechanisms to *match* the above cases, such as flow-based matching [7] and context-sensitive matching [9]. Given a rule that is similar to the rule `rButton`, these approaches can identify statements that may produce the same sequence of invocations at runtime, and return these statements as output. For example, in the first program in Figure 2, these approaches can identify the match between the rule `rButton` and the two statements in lines 2,4.

However, safely transforming these matched statements is challenging: contexts for matched statements should be considered and thus it is difficult to transform them uniformly using a simple strategy. For example, a common strategy used in existing approaches is to specify which statements in the '+' block replace which statements in the '-' block[1]. In this example, we can specify either the generated statement to replace the first statement marked with '-' or replace the second one. However, neither strategy can correctly handle even a simple case like Case-2 in Figure 2. Replacing the first statement (left program below) will

[1] This strategy can be implemented as context-sensitive rules in TXL [9], dynamic rules in Stratego [6], and standard placement of '+' statements in SmPL [27].

lead to the problem that the `parent` used in the first statement (line 2) is either uninitialized or captures some undesirable value defined by the previous context. Similarly, replacing the second statement will lead to the fact that the statement `jb.getUIClassID()` refers to a variable `jb` undefined or defined by some undesirable object in the previous context.

```
1 //jb = new JButton();
2 jb = new Button(parent, SWT.PUSH);
3 /* other rules will transform the
      following two statements */
4 s = jb.getUIClassID(();
5 parent = new JPanel();
6 //parent.add(jb);
```

```
1 //jb = new JButton();
2 /* other rules will transform the
      following two statements */
3 s = jb.getUIClassID(();
4 parent = new JPanel();
5 //parent.add(jb);
6 jb = new Button(parent, SWT.PUSH);
```

To correctly handle such cases in existing languages [27, 6, 9], we have to write rules separately for different cases, capturing all possible contexts for all transformations that need to be performed. This adds a lot of burdens to the developers, and also there is no guarantee on whether these more complex patterns have the same meaning expressed in the basic rule.

In this paper, we present a novel transformation technique, *guided normalization*, which extends the notion of syntactical transformation expressed with a simple patch rule into a more expressive transformational semantics that can *match* and *transform* statements in control flow graphs. Using our approach, the developer only has to specify the transformation rules for consecutive blocks. The system first finds matched statements in a way similar to existing approaches, and for code pieces that cannot be directly transformed, a guided normalization process is performed: the system performs a set of semantics-preserving transformations to the program so that the matched statements become a consecutive block and can be directly transformed by simply syntactic substitution. If the normalization procedure cannot be found by the system, a warning is generated to the user.

In this paper, we consider several semantics-preserving transformation primitives that are verified and widely used in compiler optimization [37, 1, 8, 14] and refactoring [33, 26, 31], including (1) *shifting* unconditional statements into all conditional branches when data dependency[2] does not break, (2) *swapping* two adjacent statements that have no data dependency with each other, and (3) *renaming* variables to their must-aliases or fresh names. Applying the three primitives on the three cases above, we can get the following programs, all of which can be directly matched and transformed by `rButton`.

```
//Case 1':
if(parent != null) {
  jb = new JButton();
  parent.add(jb);
}
```

```
//Case 2':
parent = new JPanel();
jb = new JButton();
parent.add(jb);
s=jb.getUIClassID(();
```

```
//Case 3':
jb = new JButton();
parent.set(jb);
defaultButton = jb;
```

A particular difficulty in designing guided-normalization algorithm is that several match instances can intertwine with each other, thus, a global solution is necessary to ensure that the normalization guided by one rule would not prevent the normalization guided by another rule. In our approach, we globally consider constraints brought by matches and dependencies, and then normalize the program without breaking any of them.

Challenge 2. With guided normalization, we can ensure that code pieces in different forms are transformed as a consecutive block. However, the basic transformation specified by the user rules may also lead to undesirable conflicts with other code pieces. Here we focus on one type of such conflicts: breaking def-use relation.

[2] Note there are three types of data dependencies: read-after-write, write-after-read, and write-after-write. In this paper, when we use the term "data dependency", we refer to all the three.

An example of breaking def-use relation is shown below.

```
1 //BadRule:
2 (x: ClassX->ClassX2,
3  y: ClassY->ClassY2) {
4   - x = new ClassX();
5   - y.add(x);
6   + y = new ClassY2(x);
7 }
```

```
1 //Program
2 y = System.defaultY();
3 x = new ClassX();
4 y.add(x);
5 SomeUse(y);
6 SomeOtherUse(x);
```

BadRule →

```
1 //Transformed Program
2 y = System.defaultY();
3 y = new ClassY2(x);
4 SomeUse(y);
5 SomeOtherUse(x);
```

As we can see, BadRule breaks the def-use relation in the program. The value `y` used in `SomeUse(y)` was defined by line 2 in the original program (left) and now is defined by line 3 in the transformed program (right). Similarly, the value of `x` in `SomeOtherUse(x)` was defined in the old program, and becomes undefined after the transformation. This problem is caused by the definition of BadRule, where a definition of `y` is undesirably added and a definition to `x` is undesirably removed.

We solve this problem by providing a static checker to check the def-use safety by looking into the set of transformation rules. Though this restricts the forms of the patch patterns to write, we will show in the evaluation section that such restriction will not damage our expressiveness in expressing API transformation problem.

Contributions. Concretely, this paper makes the following contributions.

- We propose a novel transformation technique, *guided normalization*, for program transformation with many-to-many mappings. In our approach, the developers specify the transformations only for consecutive method invocations, and the system automatically applies the transformation to many different forms via guided normalization.
- We design a patch-rule like programming language, PATL, to transform programs between APIs, and provide a static checker to ensure block-level transformation safety in terms of def-use relation. We formalize the semantics of core PATL atop of Middleweight Java [5], and prove (1) the guided normalization is semantics-preserving, and (2) the transformation never breaks def-use relations.
- We have evaluated our language with three non-trivial case studies for Google Calendar API updates, switching from JDom to Dom4j, and switching from Swing to SWT. The evaluation shows that our language allows successful transformations of real world programs with a small number of rules, requiring only a small amount of manual resolutions.

In the rest of the paper, we first present the syntax and basic semantics of PATL (Sections 2, 3). Then we describe how we address the two challenges (Sections 4, 5). Next, we present the implementation (Section 6) and evaluation (Section 7). Finally, we discuss related work (Section 8) and conclude the paper (Section 9).

2 PATL Syntax

2.1 Background: Middleweight Java

Our discussion of PATL is based on a formal imperative core of the Java language, Middleweight Java (MJ) [5]. To simplify the formal presentation, we only consider MJ in three-address form, and in Section 6 we shall discuss how to transform a program into and out of three-address form.

In three-address MJ, arguments of method invocations or object constructions, field access targets, `if` condition expressions and `while` condition expressions are limited to variables.

p	::=	$\overline{cd}$	(Program)
cd	::=	`class` C `extends` C $\{\overline{fd}\ cnd\ \overline{md}\}$	(Class Definition)
fd	::=	$C\ f$	(Field Definition)
cnd	::=	$C(\bar{C}\ \bar{x})\{$`super`$(\bar{e});\ vd;\ \bar{s}\}$	(Constructor)
md	::=	$\tau\ m(\bar{C}\ \bar{x})\{vd;\ \bar{s}$ `return` $x;\}$	(Method Definition)
τ	::=	`void` $\mid C$	(Return Types)
vd	::=	$\bar{C}\ \bar{x}$	(Variable Declaration)
s	::=	ps	(Statement)
	$\mid$	`if`$(x)\ \{\bar{s}\}$ `else` $\{\bar{s}\}$	
	$\mid$	`while`$(x)\{\bar{s}\}$	
ps	::=	$x = e;\ \mid x.f = e;\ \mid pe;$	(Primitive Statement)
e	::=	`null` $\mid x \mid x.f \mid (C)\ x \mid pe$	(Expression)
pe	::=	$x.m(\bar{x})$ $\mid$ `new` $C(\bar{x})$	(Promotable Expression)

Figure 3 Syntax of three-address MJ, where x ranges over MJ variables, f ranges over field names, C ranges over class names, and m ranges over method names.

Π	::=	$\pi_1, ..., \pi_n$	(Rule Sequence)
π	::=	$(\bar{d})\ \{I^-; I^+\}$	(Transformation Rule)
d	::=	$u : C \hookrightarrow C$	(Metavariable Declarations)
I^-	::=	$-\ p_1, ..., -\ p_n$	(Source Pattern)
I^+	::=	$+\ p_1, ..., +\ p_n$	(Target Pattern)
p	::=	$u = r;\ \mid r;$	(Statement Pattern)
r	::=	$u.m(\bar{u})$ $\mid$ `new` $C(\bar{u})$ $\mid u.f$	(Expression Pattern)

Figure 4 PATL syntax, where u ranges over PATL metavariables and C ranges over MJ types.

Besides all local variables in a method body are declared before the statements. The syntax of three-address Middleweight Java is presented in Figure 3.

In the formal notations, we use the bar notation adopted by Pierce [29] for repetitive elements: $\bar{a}$ indicate a sequence $a_1, a_2, ..., a_n$, and all operations defined on single values expand component-wisely along with the sequence, e.g. $\bar{C}\ \bar{f}$, is equal to $C_1\ f_1, \cdots, C_n\ f_n$, where n is the length of $\bar{C}$ and $\bar{f}$.

2.2 Syntax

The syntax of PATL is formally presented in Figure 4. Similar to SmPL [27] and SWIN [18], a PATL program is a set of patch-like transformation rules. Each rule π in a PATL program consists of two parts: 1) metavariable declarations with type information of metavariables: a declaration $v : C_1 \hookrightarrow C_2$ means that the type of v in old API is C_1 and its type is C_2 after transformation, and 2) the rule body consists of a sequence of patterns. The I^- pattern block describes the statements to be deleted and the I^+ block describes the statements to be generated. In a PATL rule π, all metavaribales used in I^- and I^+ are required to be declared in the metavariable declaration part of that rule, and particularly, when a metavariable u is newly-introduced in I^+ (u does not appear in I^-, meaning that u has no source type), u should be defined with a dummy type $\perp$ as its old type in its declaration.

3 Basic Semantics

Basic semantics of PATL performs only strict match on consecutive blocks and syntactical transformation. In Section 5 we shall describe how to extend the basic semantics to deal with code blocks in different forms.

3.1 Match

Intuitively, statements $\bar{s}^-$ in method M are matched by a rule π defined in a rule set Π, if 1) $\bar{s}^-$ is a sequence of consecutive statements in M that can be matched by the source pattern $\pi.I^-$, and 2) all variable occurrences matched by a same metavariable should have a same name. The formal definition is presented below. We assume each location in a program is uniquely identified, and use the notation x^l in the rest of the paper to denote the occurrence of variable x at location l in the program.

▶ **Definition 1** (Match). Given a method M, statements $\bar{s}^-$ in M are said to form a *match instance* with a transformation rule $\pi = (\bar{d})\{I^- \ I^+\}$ if the following conditions are satisfied.

- (Block) A basic block $\bar{s}$ exists in M, s.t. $\bar{s}^-$ is a consecutive statement sequence in $\bar{s}$.
- (Source pattern match) $\bar{s}^-$ can be matched by the pattern block I^- syntactically. There exists a map ϕ from variable occurrences to metavariable names, s.t. by substituting each variable occurrence in $\bar{s}^-$ with its image in ϕ, $\bar{s}^-$ is exactly the same as I^-.
- (Variable mapping) Suppose x^l and $y^{l'}$ are two variable occurrences matched by the same metavariable u, then $x \equiv y$ (i.e. x and y are the same variable), and these two occurrences x^l, $y^{l'}$ are must aliases.
- (Variable typing) Suppose a variable occurrence x^l is matched by a metavariable u, then $\mathsf{type}(x) <: \mathsf{type}(u)$ if x^l is a right-value in M, and $\mathsf{type}(u) <: \mathsf{type}(x)$ if x^l is a left-value in M (Function $\mathsf{type}(x)$ refers to the type of x in M, and $\mathsf{type}(u)$ refers to the source type of metavariable u in π). ◀

We denote a *match instance* as a triple $b = (\bar{s}^-, \pi, \sigma)$, where $\sigma = \{u_1 \mapsto [x^{l_1}, ..., x^{l_n}], ...\}$ is the set of mappings from metavariables to variable occurrences formed in the match. An element $u_1 \mapsto [x^{l_1}, ..., x^{l_n}] \in \sigma$ indicates that the metavariable u matches to different occurrences $x^{l_1}, ..., x^{l_n}$ of the variable x in the program.

▶ **Definition 2.** A matching instance set $\mathcal{B}$ is a set of match instances $\{b_1, ..., b_n\}$ formed by matching a method M against a set of rules Π, such that for any $b_1 \neq b_2 \in \mathcal{B}$, where $b_1 = (\bar{s_1}^-, \pi_1, \sigma_1)$ and $b_2 = (\bar{s_2}^-, \pi_2, \sigma_2)$, the statements $\bar{s_1}^-$ and $\bar{s_2}^-$ should have no overlap.

The 'none-overlapping' constraint is to ensure that one statement will not be deleted twice in transformation, and a warning will be generated if two mapping instances overlap.

Note that in our pattern definition there is no syntax for capturing a field access expression. Here we simply treat field access expressions as variables and allow metavariables to match them.

Example. A match instance set is presented in Figure 5: given the source program on left of Figure 5 and the transformation rules in Figure 1, the two statements in lines 2,3 will be matched by the rule `rButton`, and the statement in line 5 can be matched by the rule `rSetAlign`, and the match instance set is presented below: (Here, s^2 refers to the statement `btn=new Button();` in line 2, and $\texttt{btn}^2$ refers to the occurrence of the variable `btn` in line 2, other notations are similar.)[3]

$$\mathcal{B} = \big\{(s^2 s^3, \texttt{rButton}, \{\texttt{jb} \mapsto \{\texttt{btn}^2, \texttt{btn}^3\}, \texttt{parent} \mapsto \{\texttt{panel}^2\}\}), \\ (s^5, \texttt{rSetAlign}, \{\texttt{jb} \mapsto \{\texttt{btn}^5\}, \texttt{align} \mapsto \{\texttt{alX}^5\}\})\big\} \quad (1)$$

[3] If there exist two variables at the same line, the encoding of l will not just be the line number but both its line number and its character offset number, but as there is no such case in this example, we only use line number for convenience.

```
1 //Program
2 btn = new JButton();
3 panel.add(btn);
4 alX = 10;
5 btn.setAlignmentX(alX);
```

rButton
rSetAlign

```
1 //Transformed Program
2 btn = new Button(panel, SWT.PUSH);
3 alX = 10;
4 btn.setAlignment(alX);
```

Figure 5 A transformation example with rules defined in Figure 1, the statements in lines 2,3 (left) are transformed into the statement in line 2 in the result program (right) by rule `rButton`. Similarly, the statement in line 5 is transformed to the statement in line 4 (right) by rule `rSetAlign`. In our examples, variables are always declared in the previous context, but we omit them for concision consideration.

3.2 Transformation

The transformation of a program based on a match instance set $\mathcal{B}$ is defined below.

▶ **Definition 3** (Target Pattern Instantiation)**.** Given a match instance set $\mathcal{B}$, suppose $b = (\bar{s}^-, \pi, \sigma) \in \mathcal{B}$, then $\pi.I^+$ will be instantiated into $\bar{s}^+$ by substituting each metavariable u in $\pi.I^+$ with an MJ variable:

- If u is a metavariable in $\pi.I^+$ and there exists $u \mapsto [x^{l_1}, ..., x^{l_n}] \in \sigma$, then u will be substituted with x.
- If u is a metavariable in $\pi.I^+$ and u is a metavariable not appearing in $\pi.I^-$, but defined in $\pi.I^+$, then u will be instantiated as a new variable name.

▶ **Definition 4** (Statement-level Transformation)**.** Given a method M (with its statement body $\bar{s}_M$) and a rule Π we denote the transformation of $\bar{s}_M$ by Π as $\Pi \triangleright \bar{s}_M$. Suppose $\mathcal{B} = \{b_1, ..., b_n\}$ is the set of match instances between M and Π where $b_i = (\bar{s}_i^-, \pi_i, \sigma_i)$, then $\Pi \triangleright M$ is the result of substituting all $\bar{s}_i^-$ with $\bar{s}_i^+$ (statements instantiated from $\pi_i.I^+$ with b_i). Formally, $\Pi \triangleright \bar{s}_M = [\bar{s}_1^- \mapsto \bar{s}_1^+, ..., \bar{s}_n^- \mapsto \bar{s}_n^+]\bar{s}_M$. ◀

Besides statement-level transformation, we will also 1) add the definitions of newly introduced variables in the variable declaration field of M (the type of a newly introduced variable is the target type of its corresponding metavariable defined in the rule) and 2) transform the types based on the type mapping information provided in metavariable declarations in each rules in Π. But as they are not our focus in this paper, we do not formalize these two transformations.

Example. Given the match instances in Equation 1 between the program in left Figure 5 and rules in Figure 1, target statements will first be generated before substitution. In the first match instance, the target statement is generated by substitution of metavariable `jb` with `button`, and substitution `parent` with `panel`. The resulting target statement is `btn=new Button(panel, SWT.PUSH);`. Similarly, by substituting `jb` with `btn` and substituting `align` with `alX`, we obtain the target statement `btn.setAlignment(alX);` from the second match instance. By substituting source statements in each match instance with its corresponding target statements, the desirable result can be obtained (right of Figure 5).

4 Preserving Def-Use Relations

As mentioned in Challenge 2 in the introduction, we would like to disallow rules that may change def-use relations. Thus, besides checking the syntax and type correctness of the

rules as SWIN [18] did, an addition set of well-formedness conditions are checked against a transformaion program Π to ensure the it will perserve def-use relations in transformation.

▶ **Definition 5** (Well-Formedness Conditions). A PATL program $\Pi = \pi_1, ..., \pi_n$ is well-formed if the following four conditions are satisfied.

- *(Definition deletion)* For all π, if there exists a pattern $-\ u = r$ in I^-, then there should also be a pattern $+\ u = r'$ in I^+.
- *(Definition introduction)* For all π, if there exists a pattern $+\ u = r$ in I^+, then either there exists $-\ u = r$ in I^- or u does not appear in I^-.
- *(New metavariable introduction)* Given a rule π, if there exists a pattern $+p$ in $\pi.I^+$ such that a metavariable u in p does not appear in $\pi.I^-$, then there exists a pattern $+u = r$ in $\pi.I^+$ that introduce the definition of u. ◀
- *(SSA pattern form)* For all π, the code block formed by I^- (and I^+) must be in static single assignment (SSA) form, i.e. each variable is assigned at most once.

The first and second conditions ensure the def-use relations of existing variables. The third condition ensures that the new variables are used after definition. The fourth condition makes the other three conditions simpler, and also contributes to guided normalization to be introduced in the next section. As an example, the BadRule in the introduction violates the first two conditions, and thus can potentially violate def-use relations.

Now we can formally define and prove the def-use preservation property of a checked PATL program Π. Suppose $\mathcal{B}$ is the match instance set between a method M and Π, $\Pi(s)$ is used to refer to the statements corresponding to s after transformation, defined as: (1) the statements generated from a binding b, if exists $b = (\bar{s}^-, \pi, \sigma) \in \mathcal{B}$ s.t. $s \in \bar{s}^-$, or (2) s itself, if s is not matched by any rule. We have the following theorem.

▶ **Theorem 6** (Def-use Preservation[4]). *Given a method M and a checked PATL program Π, suppose the statement-level transformation is $\Pi \triangleright \bar{s}_M = \bar{s}'_M$, then:*

1. *Suppose s_1 and s_n are two statements in a basic block in $\bar{s}_M$, and there exists a variable x that is used in s_n and defined in s_1, then after transformation, either $\Pi(s_n)$ does not use the variable x, or the occurrences of x used in $\Pi(s_n)$ is defined in $\Pi(s_1)$.*
2. *Suppose x is a variable newly introduced in $\bar{s}'_M$, then x is used after definition.*

5 Extending Basic Semantics via Guided Normalization

As mentioned in Challenge 1 in the introduction, we use guided normalization to extend the basic semantics to different cases. We first start with the introduction of program analysis techniques used in our approach, then introduce the extended match that identifies different cases, and show how to guided-normalize the matched statements into the basic form.

5.1 Program Analysis

In order to apply a semantics-preserving transformation to a program p, the following two program analysis results are required.

[4] Please refer to the appendix for proofs of all theorems and properties.

1. Alias relations between variable occurrences in p: given two variable occurrences x^{l_1} and y^{l_2}, identify whether x^{l_1} and y^{l_2} are *none-aliases*, *may-aliases* or *must-aliases*[5].
2. Dependency relations between statements in p: given two primitive statements s_1 and s_2, whether there are no dependencies between them or there may exist dependencies between them, considering both data dependency and control dependency.

As the transformation algorithm only requires the analysis results, the analysis algorithm is orthogonal to the transformation phase and any *conservative* program analysis tool providing these results can be used. And as we will show later, the precision of analysis result will not affect the semantics-preserving property of the transformation process due to our conservative treatment of the analysis result, but less precise analysis result may increase the number of warnings reported by the transformation algorithm that require manual resolutions.

5.2 Extending Match

Now we define the extended match, considering match instances formed by potentially scattered consecutive statements and rules.

▶ **Definition 7** (Match*). Given a method M, statements $\bar{s}^-$ in M are said to form a *match instance* with a transformation rule $\pi = (\bar{d})\{I^- \ I^+\}$ if the following conditions are satisfied.

- (Path*) There exists a statement sequence $\bar{s}$, which forms an execution path in the control flow graph of M, s.t. $\bar{s}^-$ is a (potentially scattered) sub-sequence in $\bar{s}$.
- (Source pattern match) $\bar{s}^-$ can be matched by pattern I^- syntactically.
- (Variable mapping*) Suppose x^l and $y^{l'}$ are two variable occurrences matched by the same metavariable u, then x^l and $y^{l'}$ are may-aliases in M.
- (Variable typing) Suppose a variable occurrence x^l is matched by a metavariable u, then $\mathsf{type}(x) <: \mathsf{type}(u)$ if x^l is a right value in M, and $\mathsf{type}(u) <: \mathsf{type}(x)$ if x^l is a left value in M. ◀

Different from Definition 1, in Definition 7: (1) statements are only required to appear in a path in the control flow graph, and (2) variable occurrences bound to the same metavariable are only required to be may-aliases. Here, we use $\mathcal{B}^* = \{b_1^*, ..., b_n^*\}$, where $b_i^* = (\bar{s_i^-}, \pi_i, \sigma_i^*)$, to refer to the set of match instances formed between a method M and a PATL program Π with extended match definition Match^*. The difference between σ^* and σ is that given $u \mapsto [x_1^{l_1}, ..., x_n^{l_n}] \in \sigma^*$ (indicating metavariable u maps to the variable occurrences $x_1^{l_1}, ..., x_n^{l_n}$ in the match instance), occurrences $x_1^{l_1}, ..., x_n^{l_n}$ are only required to be aliases but not necessarily with the same name.

Match Finding and Checking. Given a method M and a set of rules Π, we use a dataflow analysis to obtain all match instances between statements in M and rules in Π. The behavior of the dataflow analysis is similar to the method presented by Brunel et al. [7]. Due to space limit, we omit the details here.

As mentioned before, not all match instances can be guided-normalized. If there exist match instances that cannot be handled by guided-normalization, we will report them to users as warnings. Some untransformable match instances can be easily identified by static condition check. More will be identified during the process of guided normalization. Basically,

[5] Typical analysis tools will only provide none-alias relations and may-aliases relations, but enhancing them with a conservative intra-procedural aliases analysis can further provides must-alias relations between variable occurrences.

we identify any match instance $b^* = (\bar{s}_1, \pi, \sigma^*)$ satisfying one of the following three conditions and report it as a warning.

- Any two variable occurrences matched by the same metavariable in σ^* are not must-aliases.
- Statements $\bar{s}_1$ appear across methods.
- Statements $\bar{s}_1$ appear across boundary of a `while` statement, i.e., the matched statements are in different iterations, or, some are inside a loop while some outside.

The latter two conditions are checked after the matching process, while the first condition is checked after the guided-shift step in guided normalization (introduced later in this section). This is because the guided-shift step will eliminate some may-aliases, and after checking, all uncertain aliases between variables occurrences involved in match instances are resolved. On the other hand, other uncertain aliases relations will not need to be handled as they will not affect transformation correctness, as a result, only a small part of uncertainties is required to resolve to proceed transformation.

Example. An example program demonstrating extended match definition with rules `rButton` and `rSetAlign` is presented below on the left of Figure 6. In this program, firstly, the alignment field of the button is set via a call to "`btn.setAlignmentX(alX);`". Then, before adding the new `JButton` object to `panel`, it checks whether `panel` is `null`: if it is not null, then the `btn` is added to `panel`, otherwise the button is assigned to `defaultBtn`, added to a default panel `defaultPnl`.

From pointer analysis we can obtain that all occurrences of `btn` and `defaultBtn` are may-aliases. Thus, three match instances (in Match*) can be obtained between the program and the rules in Figure 1: i.e. 1) statements in lines 1,6 can form a match instance with the rule `rButton`, 2) statements in lines 1,9 can form another match instance with the rule `rButton` and 3) statement in line 2 can be matched by the rule `rSetAlign`. And these match instances can be represented as Equation 2:

$$\begin{aligned}\mathcal{B}^* = \{&(s^1 s^6,\ \texttt{rButton},\ \{\texttt{jb} \mapsto \{\texttt{btn}^1, \texttt{btn}^6\}, \texttt{parent} \mapsto \{\texttt{panel}^6\}\}),\\ &(s^1 s^9,\ \texttt{rButton},\ \{\texttt{jb} \mapsto \{\texttt{btn}^1, \texttt{defaultBtn}^9\}, \texttt{parent} \mapsto \{\texttt{defaultPnl}^9\}\}),\\ &(s^2,\ \texttt{rSetAlign},\ \{\texttt{jb} \mapsto \{\texttt{btn}^2\}, \texttt{align} \mapsto \{\texttt{alX}^2\}\})\}\end{aligned} \tag{2}$$

5.3 Guided Normalization

Guided normalization transforms the program in a semantics-preserving manner such that the match instance in the extended semantics can be transformed by the basic semantics. We first demonstrate the result of guided normalization by example.

Example. A desirable guided-normalization for the program with the match instances in Equation 2 is presented below (6 right), as after normalization, statements matched to a same rule appear consecutively in basic block and variable occurrences matched by a same metavariable have same name. It is obvious that after guided-normalization, the transformation defined in Section 3 can be performed as all extended match instances become the basic in-block matches. Please note in this program different API methods do not write or read to the same field, so there is no dependency between these method calls. This information can be obtained by a dependency analysis.

Semantics Preserving Transformation. As mentioned in Section 1, the key point of guided-normalization is to ensure that the normalization process is semantics-preserving. Thus before moving to an algorithmic description of guided-normalization, we shall first define which transformations are semantics-preserving.

```
1 btn = new JButton();
2 btn.setAlignmentX(alX);
3 System.out.print(alX);
4 b = panel != null;
5 if (b) {
6   panel.add(btn);
7 } else {
8   defaultBtn = btn;
9   defaultPnl.add(defaultBtn);
10 }
```

normalize →

```
1 System.out.print(alX);
2 b = panel != null;
3 if (b) {
4   btn = new JButton();
5   panel.add(btn);
6   btn.setAlignmentX(alX);
7 } else {
8   x = new JButton();
9   defaultPnl.add(x);
10  btn = x;
11  defaultBtn = btn;
12  btn.setAlignmentX(alX);
13 }
```

Figure 6 An Example Program to be Transformed by Rules `rButton` and `rSetAlign` and its guided normalized result. The program on the right is normalized with match instances in Equation 2, and after guided normalization, the match instances become: 1) statements in lines 4,5 form a match with rule `rButton`, 2) statement in line 6 form a match with rule `rSetAlign`, 3) statements in lines 8,9 form a match with rule `rButton` and 4) statement in line 12 form a match with rule `rSetAlign`.

▶ **Definition 8** (Semantics-preserving transformation)**.** A method M (the body of M is $\bar{s}_M$) is said to be semantics-equivalently transformed into M', if M can be transformed into M' with a series of the transformation primitives defined below. We denote such semantics-preserving transformation as $M \xrightarrow{\sim} M'$.

- *Alias Renaming Primitive:* In $\bar{s}_M$, if x^l is a must alias of variable y in the statement of l, renaming x to y at the statement of l is semantics-preserving.
- *Left-value Renaming Primitive:* In $\bar{s}_M$, suppose $x = e;$ is a statement defining the value of x, and y is a free name in M, then after declaring y in M, substituting $x = e;$ with $y = e; x = y;$ is semantics preserving.
- *Fresh-variable Introduction Primitive:* In $\bar{s}_M$, suppose x is a fresh variable name in M, then declaring x in M and inserting $x = y;$ at a location where y is defined is semantics-preserving.
- *Swapping Primitive:* In $\bar{s}_M$, suppose $s_1 s_2$ are two adjacent statements with no dependency, then transforming them into $s_2 s_1$ is semantics preserving.
- *Shifting Primitive:* In $\bar{s}_M$, (1) given $s_1\texttt{if}(x)\{\bar{s}_2\}\texttt{else}\{\bar{s}_3\}$ and suppose x does not depends on s_1, then substituting it with $\texttt{if}(x)\{s_1\bar{s}_2\}\texttt{else}\{s_1\bar{s}_3\}$ is semantics preserving. (2) given $\texttt{if}(x)\{\bar{s}_2\}\texttt{else}\{\bar{s}_3\}s_1$, substituting it with $\texttt{if}(x)\{\bar{s}_2 s_1\}\texttt{else}\{\bar{s}_3 s_1\}$ is semantics preserving. ◀

The transformation primitives defined above are verified and commonly used in compiler optimization [37, 1, 8, 14] and semantics-preserving refactoring [33, 26, 31], and as long as we can show that a transformation algorithm can be decomposed into such series of transformation primitives, the transformation process is guaranteed to be semantics-preserving.

Transformation Stages. Algorithmically, the guide-normalization process can be decomposed into the following three stages:

1. Stage-1: Transforming the program with GuidedShift Algorithm so that statements matched by a rule will appear in a basic block in the resulting program.
2. Stage-2: Transforming the program with GuidedRename Algorithm, and variable occurrences matched a same metavariable in a rule will have same name and definition.
3. Stage-3: Transforming statements in basic blocks with GuidedReorder Algorithm, so that statements matched by a same rule will appear consecutively in the block.

In the rest of the section, we will concretely describe each guided normalization stage and prove its semantics-preserving property. The guided normalization process consists of three transformation stages (three algorithms), i.e. GuidedShift, GuidedRename and GuidedReorder.

In our algorithms, we use Δ to denote a *must-alias* checker, which is obtained by program analysis, i.e. given two variable name x, y and a location l, $\Delta(x, y, l) = \texttt{true}$ indicates that x and y are must-aliases at the location l otherwise not. We also use Θ to represent a dependency checker, i.e. given two statements, $\Theta(s_1, s_2) = \texttt{true}$ indicates that s_1 and s_2 may have dependencies otherwise not [6]. Particularly, besides dependencies obtained from analysis, here we also consider *match dependency*: two statements s_1 and s_2 are said to have match dependency when they are in a same match instance in $\mathcal{B}^*$. We shall refer match dependency and data dependency uniformly as dependency.

5.3.1 GuidedShift Algorithm

In the first stage, we make all statement sequences in a match instance appear in the same basic block, i.e. given the match instance set $\mathcal{B}^* = \{(\bar{s}_1^-, \sigma_1^*, \pi_1), ...\}$, if $\bar{s}'_M = \mathsf{GuidedShift}(\bar{s}_M, ST_{\mathcal{B}^*}, \Theta)$, then all $\bar{s}_i^-$ in match instances from $\mathcal{B}^*$ will appear in a basic block in $\bar{s}'_M$. Here $ST_{\mathcal{B}^*}$ is the shifting targets of $\mathcal{B}^*$, indicating which statements are supposed to appear in a basic block. $ST_{\mathcal{B}^*}$ is calculated by including all adjacent statement pairs in $\bar{s}_i^-$, formally, given $b^* = (\bar{s}_i^-, \sigma_i^*, \pi_i) \in \mathcal{B}^*$, if $s_1 s_2$ are two adjacent statements in $\bar{s}_i^-$, then a pair (s_1, s_2) is added into $ST_{\mathcal{B}^*}$.

The function $\mathsf{locateBlock}(s)$ is used to find the basic block where s is in, ShiftDownInto (ShiftUpInto) is used to move a statement into the beginning (end) of both branches of an `if` statement without moving any other statement, and UpdateLocation is used to update locations for all statements whenever a shift operation happens (as it changes locations of statements).

In the algorithm, given a target pair (s_a, s_b), we will first find two compound statements s_1, s_2 such that s_1, s_2 appear in the same basic block and s_1 contains or equals s_a, s_2 contains or equals s_b (line 2). Then we determine whether s_1 is a compound statement containing s_a. If so, we shift s_2 into an inner level of the block s_1 (lines 3-13). In the shifting process, we visit each statement after s_1 in the block s_1 is in, and when a statement s' having dependence with s_2 is found, it will be shifted first to avoid dependency breaking (lines 8-10). If no such statement exists, s_2 will moved into an inner level of the block s_1 using the shiftUpInto function (line 8-10). When $s_1 = s_a$, meaning that s_a is in an outer block level compared to s_b, we need to shift s_a into s_2 unless s_a and s_b are already in the same basic block (lines 14-28). This process is similar to the former part except that dependence relation will be checked when we try to shift a statement into an if statement, if dependence between the condition variable and the statement exists, an warning will be generated for user to handle (line 24).

The algorithm will always terminate as the block-level (the number of nested blocks a statement is in) of some statement will increase in each loop. And the algorithm only terminates when the shifting goal is satisfied. The semantics-preserving property of the algorithm is presented below.

▶ **Property 9** (GuidedShift semantics-preserving). *Let M be a method ($\bar{s}_M$ be its body), Θ be a dependency checker that contains all statement dependencies in $\bar{s}_M$, and ST be a*

[6] If may-dependence relation is reported between two statements s_1 and s_2, the result will be treated as "s_1 and s_2 have dependencies" in our approach. This treatment ensures that any *possible* dependencies will not be broken in transformation.

Algorithm 1: GuidedShift algorithm.

Input: statements $\bar{s}$, shifting targets $ST = [(s_a, s_b), ...]$,
dependency checker Θ.
Output: shifted statement sequence $\bar{s}'$

while *exists tuple* $(s_a, s_b) \in ST$, *s.t.* s_a *and* s_b *are not in the same block* **do**
  find s_1, s_2, *s.t.* s_1 *is the statement containing* s_a, s_2 *is the statement containing* s_b *and* s_1, s_2 *are in the same basic block.*
  if $s_1 \neq s_a$ ***and*** s_1 *is a compound statement containing* s_a **then**
    $\bar{s}_t \leftarrow \mathsf{locateBlock}(s_1)$;
    $i_1 \leftarrow \mathsf{indexOf}(s_1, \bar{s}_t)$;
    for $i \leftarrow i_1 + 1, ..., \mathsf{size}(\bar{s}_t) - 1$, **do**
      $s_x \leftarrow \bar{s}_t[i]$
      if $s_x == s_2$ **then**
        $\mathsf{shiftUpInto}(s_2, s_1)$; $\mathsf{updateLocations}(\bar{s})$;
        break;
      if $\Theta(s_2, s_x) ==$ `true` **then**
        $\mathsf{GuidedShift}(\bar{s}_t, [(s_x, s_1)])$;$\mathsf{updateLocations}(\bar{s})$
        break;
  else if $s_1 = s_a$ ***and*** $s_2 =$ `if`$(u)\{s_{21}\}$`else`$\{s_{22}\}$ **then**
    $\bar{s}_t \leftarrow \mathsf{locateBlock}(s_a)$;
    $i_2 \leftarrow \mathsf{indexOf}(s_2, \bar{s}_t)$;
    for $i \leftarrow i_2 - 1, ..., 0$ **do**
      $s_x \leftarrow \bar{s}_t[i]$
      if $s_x == s_a$ **then**
        if $\Theta(u, s_a) ==$ `false` **then**
          $\mathsf{shiftDownInto}(s_a, s_2)$; $\mathsf{updateLocations}(\bar{s})$;
          break;
        else
          $\mathsf{report}()$; $\mathsf{retract}()$; $\mathsf{exit}()$;
      if $\Theta(s_x, s_1) ==$ `true` **then**
        $\bar{s}_t \leftarrow \mathsf{GuidedShift}(\bar{s}_t, [(s_x, s_2)])$;
        $\mathsf{updateLocations}(\bar{s})$
        break;

set of shifting targets. We have $M \xrightarrow{\sim} [\bar{s}_M \mapsto \bar{s}'_M]M$ *if the invocation to the algorithm* $\mathsf{GuidedShift}(\bar{s}_M, ST, \Theta) = \bar{s}'_M$ *finishes without warning.*

Example. Given the example in Figure 6, the first stage transformation is shifting, and it is performed as follows in Figure 7.

5.3.2 GuidedRename Algorithm

The second stage of guided-normalization is to deal with variable names in $\bar{s}_M$, so that if there exist two variable occurrences x^{l_1}, y^{l_2} matched to the same metavariable u in a match instance (x^{l_1}, y^{l_2} are aliases to form such match), we will rename them into a uniform name via semantics-preserving transformation. Since a variable may be re-assigned or used outside a matching instance, to avoid disturbance to the code outside a matching instance, we always introduce a new variable and renaming to the new variable.

Similar to the previous normalization stage, the renaming targets $RT_{\mathcal{B}^*}$ are calculated first based on the $\mathcal{B}^*$. Given a match instance $b^* = (\bar{s}^-, \pi, \sigma^*) \in \mathcal{B}^*$ and $u \mapsto [x_1^{l_1}, ..., x_n^{l_n}] \in \sigma^*$, we add $[x_1^{l_1}, ..., x_n^{l_n}]$ into the renaming targets, as these variable occurrences are matched by the same metavariable u in b^*. When we obtain $RT_{\mathcal{B}^*}$ from $\mathcal{B}^*$, we will run GuidedRename on $RT_{\mathcal{B}^*}$ and M to obtain the desirable normalized program.

In our algorithm, we use the following auxiliary functions: 1) $\mathsf{Occurs}(RT)$ calculates the set of all variable occurrences appearing in RT, 2) $\mathsf{FreshName}(M)$ generates a fresh variable

```
1 btn = new JButton();
2 btn.setAlignmentX(alX);
3 System.out.print(alX);
4 b = panel != null;
5 if (b) {
6   panel.add(btn);
7 } else {
8   defaultBtn = btn;
9   defaultPnl.add(defaultBtn);
10 }
```

GuidedShift →

```
1 System.out.print(alX);
2 b = panel != null;
3 if (b) {
4   btn = new JButton();
5   btn.setAlignmentX(alX);
6   panel.add(btn);
7 } else {
8   btn = new JButton();
9   btn.setAlignmentX(alX);
10  defaultBtn = btn;
11  defaultPnl.add(defaultBtn);
12 }
```

Figure 7 Guided-shift result of the example in Figure 6 left. Based on the match instances defined in Equation 2, the statement in line 1 is the target statement to be shifted into branches. Besides, the statement in line 2 is also shifted into branches as it depends on the statement in line 1.

```
1 System.out.print(alX);
2 b = panel != null;
3 if (b) {
4   btn = new JButton();
5   btn.setAlignmentX(alX);
6   panel.add(btn);
7 } else {
8   btn = new JButton();
9   btn.setAlignmentX(alX);
10  defaultBtn = btn;
11  defaultPnl.add(defaultBtn);
12 }
```

GuidedRename →

```
1 System.out.print(alX);
2 b = panel != null;
3 if (b) {
4   btn = new JButton();
5   btn.setAlignmentX(alX);
6   panel.add(btn);
7 } else {
8   x = new JButton();
9   btn = x;
10  btn.setAlignmentX(alX);
11  defaultBtn = btn;
12  defaultPnl.add(x);
13 }
```

Figure 8 Guided-rename result following the previous result in Figure 7, a new variable name `x` is introduced to rename `btn`[8] and `defaultBtn`[11].

name that is not used in M, 3) UpdateAlias() updates variable locations in both M and RT after transformation and 4) rename(x, y, l) renames x into name y at the location l.

In GuidedRename algorithm, we will deal with left-value renaming in lines 1-9 and deal with right-value renaming in lines 10-21.

Left-value renaming (line 1-9) is used to deal with situations that the first element $x_1^{l_1}$ in a renaming target $rt = [x_1^{l_1}, ..., x_n^{l_n}]$ is a left-value in M. (As we can prove, the only possible left-value in a rt is the first element, as the patterns are required to be in SSA form and all $x_i^{l_i}$ are matched by a same metavariable u in a rule.) Firstly, we will find the statement that defines the value of $x_1^{l_1}$ and collect all such statements in a set S_1 (line 1). And then, for each collected assignment statement, we will generate a fresh name to rename it through the *Left-value renaming primitive* defined in Definition 8 (lines 2-5). Then, as all these $x_i^{l_i}$ appear in a same basic block (as a result of the GuidedShift algorithm), we will rename all variable occurrences in rt into the new name (lines 7-8). Similarly, Right-value renaming (lines 10-23) helps renaming a target such that all variable occurrences are right values.

Example. The guided-renaming phase for the program in Figure 7 is shown below in Figure 8. As in the match instance between statements in lines 8-11 and the rule `rButton` involves an alias-pair (i.e. `btn`[8] and `defaultBtn`[11]), they will be renamed to have the same name. The renaming process is done by introducing a new variable name `x` following the left-value renaming part defined in the GuidedRename algorithm (lines 8,9,12 in the right of Figure 8).

Algorithm 2: GuidedRename algorithm.

Input: method $M = \tau\ m(\bar{C}\ \bar{x})\{vd;\ \bar{s}_M;\ \texttt{return}\ x;\}$,
renaming targets $RT = \{[x_{11}^{l_{11}}, ..., x_{1n_1}^{l_{1n_1}}], ...\}$
Output: method M with some variables in the body renamed

```
1  S_1 ← Assignment statements with left-value in Occurs(RT);
2  foreach s_d ∈ S_1 (s_d of form y^l = e;) do
3      x_1 ← FreshName(M);
4      M.vd ← M.vd ∪ {type(y^l) x_1};
5      M.s̄_M ← [s_d ↦ x_1 = e; y = x_1;]M.s̄_M;
6      UpdateAlias();
7      foreach y^{l'} where y^{l'} and y^l are in a rt ∈ RT and l ≠ l' do
8          rename(y, x_1, l'); UpdateAlias();
9  S_2 ← ∅;
10 foreach rt = [x_{i1}^{l_{i1}}, ..., x_{in_i}^{l_{in_i}}] ∈ RT do
11     if Foreach x_{ij}^{l_{ij}} ∈ rt, x_{ij}^{l_{ij}} is a right-value then
12         S_2 ← S_2 ∪ {statement use x_{i1}^{l_{i1}}};
13 foreach s ∈ S_2 do
14     foreach variable y^l used in s do
15         if y^l ∈ Occurs(RT) and y^l is the first element of a rt ∈ RT then
16             x_1 ← FreshName(M);
17             M.vd ← M.vd ∪ {type(y^l) x_1};
18             M.s̄_M ← [s ↦ (u = y; [y ↦ x_1]s;)]M.s̄_M;
19             UpdateAlias();
20             foreach other y^{l'} in rt do
21                 rename(y, x_1, l'); UpdateAlias();
22 return M;
```

Similar to GuidedShift algorithm, GuidedRename is also semantics preserving, and we present the proof below.

▶ **Property 10** (GuidedRename Semantics-preserving)**.** *Let M be a method, Π be a transformation program, $\mathcal{B}^*$ be a set of match instances between M and Π, and $RT_{\mathcal{B}^*}$ be a set of renaming targets generated from $\mathcal{B}^*$. We have $M \xrightarrow{\sim} M'$ if* GuidedRename$(M, RT_{\mathcal{B}^*}) = M'$.

5.3.3 GuidedReorder Algorithm

The last phase of guided normalization is *guided-reordering*, in which we want to reorder statements in blocks so that given a match instance $b^* = (\bar{s}^-, \pi, \sigma^*) \in \mathcal{B}^*$, statements $\bar{s}^-$, will appear consecutively.

The reordering targets OT required by the algorithm will first be built: starting from an empty set, for each $b_i^* = (\bar{s_i}^-, \pi_i, \sigma_i^*) \in \mathcal{B}^*$, $\bar{s_i}^-$ will be added into OT, i.e. the goal is to make all such statement sequences matched by source patterns appeared consecutively. The dependency checker Θ here is same as the checker used in GuidedShift.

In the GuidedReorder algorithm, firstly, for each block $\bar{s}$, we assign each statement $s \in \bar{s}$ with a field l indicating its target location. Then, we encode the goal as constraints using by these locations, and then these locations can be calculated by solving the constraints. Concretely, the constraints are built in the following way:

- For each $s_i, s_j \in \bar{s}$, if $\Theta(s_i, s_j) =$ `true`, add $s_i.l < s_j.l$ into the constraint set if $i < j$, otherwise add $s_j.l < s_i.l$ if $j < i$ (line 6-10). These constraints ensure that the statement dependencies are kept after reordering.
- For each statement sequence $\bar{s}_k \in OT$, add $s_{k(i)}.l + 1 = s_{k(i+1)}$ into the constraint set (line 11-13). These constraints ensure that target statements will appear consecutively in the right order.

Algorithm 3: Statements Reordering.

```
Input: method M, dependency checker Θ,
       reordering targets OT = [s̄_a, ...]
Output: Re-ordered statement sequence s̄'
foreach basic block s̄ in M do
    OT_s ← {statement sequences from OT and in s̄};
    (Suppose OT_s is {s̄_1, ..., s̄_m})
    n ← length(s̄);
    Constraints ← ∅;
    foreach s_i ≠ s_j ∈ s̄ do
        if Θ(s_i, s_j) and i < j then
            Constraint ← Constraint ∪ {s_i.l + 1 ≤ s_j.l};
        else if Θ(s_i, s_j) and j < i then
            Constraint ← Constraint ∪ {s_j.l + 1 ≤ s_i.l};
    foreach s̄_k ∈ OT do
        foreach i = 1 to length(s̄_k) − 1 do
            Constraint ← Constraint ∪ {s_i.l + 1 = s_{i+1}.l};
    if TrySolve(Constraints) successful then
        s̄' ← Sort s̄ according to s.l;
        M ← [s̄ ↦ s̄']M;
    else
        report();
return M;
```

As these constraints form a *system of difference constraints* [10], and we can solve it through shortest path algorithm. When we successfully solve the constraints, we will then reorder the statements accordingly in M to obtain the desirable result. If there exists no solution to the constraints, warnings will be generated to users, as we cannot make all matched statements appear consecutively due to dependency issues.

Example. After performing GuidedRename the program in Figure 8, we only need to reorder the statements in lines 4,6 and statements in lines 8,12 in program (right of Figure 8) to obtain the desired program in Figure 6 left. At this point, we successfully guided-normalize the program as presented in 6.

The following property shows that GuidedReorder is semantics-preserving.

▶ **Property 11** (GuidedReorder semantics-preserving). *Let M be a method, Θ be a dependency checker containing all statement dependencies in M, and OT be a set of reordering target. We have $M \xrightarrow{\sim} M'$ if $M' =$ GuidedReorder(M, Θ, OT).*

5.4 Main Theorem

Here we present the main theorem on the semantics-preservation of guided normalization.

▶ **Theorem 12** (Main Theorem). *Let M be a method, Π be a rule set, and $\mathcal{B}^*$ be a set of match instances between statements in M and rules in Π. We have $M \xrightarrow{\sim} M'$ if the guided normalization of M with $\mathcal{B}^*$ returns M'. In other words, the transformation is semantics-preserving.*

Proof. This theorem is a simple derivation of Property 9, 10 and 11. ◀

6 Full Language Implementation

Based on the core PATL presented in previous sections, we implemented a version of PATL for Java (Patl4J). On top of core PATL, there are several extensions in the full language.

First, the full language supports standard Java programs that are not necessarily in three-address code. Before matching and transformation, we first convert the Java program into three-address code. This is achieved by introducing a set of temporary variables to decompose statements that are not in three-address form. We give special names to the temporary variables, and after the transformation, we try to apply the "inline variable" refactoring to inline these variables to recover the original program. In this way we can ensure the structure of the original program is retained to some degree.

Second, the full language supports context-sensitive matching, similar to the context-sensitive matching in SmPL [27] and TXL [9]. We introduce a new type of pattern to match a context of a rule. A context is a sequence of statements (not necessarily consecutive) that always appear before the matched statements in all paths in the control flow graph, and transformation only when its context is matched by the rule. An example of the context-sensitive matching can be found in the evaluation section.

Finally, the full language provides a new type of transformation rules to change method definitions. In Java, we may define a class that extends a library class or implements a library interface. When the library class/interface is mapped to a new class/interface, the client definition should also be changed. Our rule works similarly to refactorings [13], allowing to rename a method, reorder the parameters of a method, or introduce a new parameter.

Currently, the full language is implemented as an Eclipse Plug-in using the Eclipse JDT parser to manipulate the syntax tree and obtain type and use Soot [35] to perform program analysis for the client program. Concretely, in our implementation, desired program analysis results are obtained using SOOT Spark pointer analysis tool, where 1) the alias relation between two variable occurrences is determined by querying the anaylsis tool whether two variable occurrences always points to the same memory location and 2) the dependence relations between two statements is determined by querying the tool whether there exists variable occurrences in the two statements accessing the same memory location. Our prototype can be found in its web site[7] [8].

7 Evaluation

How effective is PATL for transforming real-world industrial cases? To answer this question, we evaluate our approach on three groups of widely-used Java API cases, i.e. Jdom to Dom4j, Google Calendar version 2 to version 3, and Swing to SWT.

7.1 Data Set

In our experiments, we chose three case studies that migrate programs from Jdom[9] to Dom4j[10], from Google Calendar[11] version 2 to version 3, and from Swing[12] to SWT[13], respectively. Jdom and Dom4j are two popular XML parsers, but Dom4j has better performance over

[7] https://github.com/Mestway/Patl4J.
[8] As a proof-of-concept, the type analysis, interprocedural dependency analysis, and some transformation steps in our implementation are not fully automated and require user inputs. Nevertheless, this is purely a pragmatic problem due to our limited resource on implementation; our approach can be implemented fully automatically.
[9] http://www.jdom.org/
[10] http://www.dom4j.org/
[11] https://developers.google.com/google-apps/calendar/
[12] http://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html
[13] http://www.eclipse.org/swt/

Table 1 Subject Client Programs.

Client	KLOC	Classes	Methods	Case
husacct	195.6	1187	5977	Jdom/Dom4j
serenoa	12.2	52	523	Jdom/Dom4j
openfuxml	112.5	727	4098	Jdom/Dom4j
clinicaweb	3.9	74	213	Calendar
blasd	9.7	199	729	Calendar
goofs	8.6	78	643	Calendar
evochamber	12.8	132	868	Swing/SWT
swingheat	2.3	30	186	Swing/SWT
marble	1.6	10	56	Swing/SWT
Total	359.2	2489	13293	–

Jdom on a number of tasks, so it is desirable to migrate programs from Jdom to Dom4j. Google Calendar API is a web service provided by Google to access personal calendar data. The interface for version 2 has been shut down, and version 3 is not compatible with version 2 clients, so the clients will not work unless migrated to version 3. Swing and SWT are two Java GUI libraries. Swing uses platform-independent components while SWT is designed as a light-weight wrapper of native GUI. SWT is sometimes considered faster than Swing, and some platforms such as Eclipse only supports SWT. We chose the three case studies because they are real-world program migration cases, and a large number of clients are available for the evaluation. Also, the three cases cover the two main types of program migration: API switching and API upgrading.

To evaluate the transformation program we wrote, we also collected a number of client programs that use the source API. These clients are obtained by searching the source API methods in searchcode[14]. In total, we used nine client programs in our evaluation. The statistics of the subject programs can be found in Table 1. In total, the projects totally contain 342.5 KLOC, including 2317 classes and 12183 methods, details about these projects can be found in our implementation website.

7.2 Procedures

For each case, we first wrote PATL rules that capture the correspondence between the old and the new APIs. Since the changed portion of the API is large, and it is important to test the rule under a real client, we only dealt with the portion of API that is used in our subject client programs. However, the rules we wrote are generic rules for any possible client, not specific to the subjects in our evaluation.

Next we transformed client programs that contain source API invocations using our transformation tools. To produce working clients, we manually resolved warnings reported during the transformation. A few API invocations are impossible to be transformed due to the limitation of PATL, and we create mock objects for them.

To ensure that the transformation is performed correctly, we performed three different tests for these converted clients. (1) Client EvolutionChamber comes with a set of 28 functional and 4 performance tests, and we ensure that the transformed client passes all

[14] http://searchcode.com

Table 2 Transformation Rules.

Transformation	Rules	Classes	Methods	M-to-m
Jdom/Dom4j	84	12	77	12(14.3%)
Calendar	42	14	45	21(50.0%)
Swing/SWT	110	40	82	54(49.1%)
Total	236	66	204	87(36.9%)

Table 3 Results of the Transformations.

Client	CF	CL	W	U	I	MM	GN
husacct	42	852(100%)	0(0%)	0(0%)	0	0(0%)	0(0%)
serenoa	8	273(98.9%)	0(0%)	3(1.1%)	0	9(3.3%)	0(0%)
openfuxml	72	983(94.8%)	0(0%)	54(5.2%)	15	2(0.2%)	0(0%)
clinicaweb	5	81(100%)	0(0%)	0(0%)	8	34(42%)	0(0%)
blasd	5	26(63.4%)	8(19.5%)	7(17.1%)	0	13(50%)	2(15.4%)
goofs	13	100(80.0%)	12(9.6%)	13(10.4%)	27	27(27%)	0(0%)
evochamber	9	587(98.3%)	10(1.7%)	0(0%)	0	330(56.2%)	109(33.0%)
swingheat	21	653(100%)	0(0%)	0(0%)	0	461(70.6%)	394(85.5%)
marble	6	488(98.6%)	0(0%)	7(1.4%)	0	240(49.2%)	220(91.7%)
Total	181	4043(97.3%)	30(0.7%)	84(2.0%)	50	1116(27.6%)	725(65.0%)

CF = number of changed files, CL= number of changed lines, percentages in CL = CL / (CL+W+U), W = the number of lines of code that have warnings, percentages in W = W / (CL+W+U), U = number of lines that PATL cannot transform, percentages in U = U / (CL+W+U), I = number of lines impossible to transform, MM = number of lines that are involved in many-to-many mappings, percentages in MM = MM / CL, GN = number of lines that require guided normalization, percentages in GN = GN / MM.

transformed tests. (2) We check whether they behave normally without exceptions, error messages, or crashes.. (3) For the clients in JDom/Dom4j and Swing/SWT, we side-by-side executed both the original and transformed clients to ensure they behaved the same. Note that we cannot apply the last test to the clients of Google Calendar because Calendar API v2 is already shut down.

7.3 Results

Rules Written. The statistics for the rules and transformed source APIs are summarized in Table 2: the 'Rule' column shows the number of the rules. The 'Class' and the 'Method' columns contian the number of API classes and methods covered by the rules. In total, we wrote 236 rules for the three case studies, and 66 classes and 204 methods in the source API are covered.

Basically, the number of rules is close to the number of covered methods. Furthermore, most rules are easy to write: only 14 rules in total that have a body longer than 4 lines. This indicates that PATL rules are friendly to users: they only need to capture basic forms in writing the rules without worrying about complex program context.

Effectiveness of Client Transformation. To evaluate the effectiveness of our approach on transforming clients, we counted the following numbers in our evaluation: (1) the number of files transformed by the rules (CF), (2) the number of lines transformed by the rules (CL), (3) the number of lines in the source client code on which warnings are generated (W), (4)

```
// Rule-1
(x: CalendarEntry->CalendarListEntry
    ,
 y: TextConstruct->String,
 z : String -> String){
    m y = new PlainTextConstruct(z);
    - x.setTitle(y);
    + x.setSummary(z);
}
```

```
//Rule-2
(y: TextConstruct -> String){
    - y = new PlainTextConstruct(z
        );
    + y = "";
}
```

Figure 9 Two transformation rules used in transforming Google Calendar v2 to v3.

the number of lines where transformed code exists but the transformation rule cannot be specified in PATL (U). Details of the data are presented in the first five columns in Table 3.

As we can see from the table, the 236 rules changed in total 4043 lines distributed in 181 files. This result reflects the effort saved from manual migration. The efforts saved are twofold: (1) to locate these lines from different files, (2) to derive transformation solution for each line based on the context surrounding this line.

Furthermore, the number of lines transformed by PATL consist of 97.3% of the lines that need to be converted. The rest of lines need manual resolution: the lines where warnings are reported and the unconvertible lines. These lines amount to 114 lines, consisting of 2.7% of total lines. This result indicates that PATL is able to handle the majority of the cases in practice.

Many-to-Many Rules. A significant portion of the transformation rules are many-to-many rules. As we can see from the last column of Table 2, in total 36.9% of the rules are many-to-many rules, and in the case of Google Calendar upgrade the percentage is as high as 50%. These many-to-many rules is responsible to transform a significant portion of the client code. As shown in column "MM" in Table 3, in total 27.6% of the lines are transformed by many-to-many rules, and in project swingheat the percentage is as high as 70.6%.

From our running example we have seen a typical case where many-to-many rules can be applied. Here we show another typical case: a wrapper class in the source API does not exist in the target API. For example, in the old version of Google Calendar, a wrapper class called `PlainTextConstruct` is used to wrap a string. For example, client goofs contains the following two lines of code (`e` in the statement is a variable of type `CalendarEventEntry`).

```
e.setTitle(new PlainTextConstruct(name));
```

In the new API, the use of the wrapper class is removed as many as possible. As a result, the above two lines should be converted to the following line of code.

```
e.setSummary(name);
```

This instance is a typical many-to-many mapping instance, and the transformation can be captured by the two rules in Figure 9.

The first rule uses context-sensitive matching described in Section 6. The statement annotated with "m" is a context pattern, which indicates that `x.setTitle(y)` is only matched and transformed if there is `y = new PlainTextConstruct(z)` before it. The first rule captures the wrapping of a string and the use of the wrapper, and converts it into a proper method invocation based on how the wrapper is used. The second rule removes the useless variable `y`. Note that the target method `setSummary` is decided by both the argument passed to `PlainTextConstruct` (in this case, a string but not an html object) and the invoked source method on `x` (in this case `setTitle`), so this rule has to be many-to-many.

```
XMLOutputter out = new XMLOutputter();
for (Element e : rulesElements)
  rulesToRegister += out.outputString(e);
```

```
StringWriter sw = new StringWriter();
XMLWriter out = new XMLWriter(sw);
for (Element e : rulesElements)
  out.write(e);
rulesToRegister += sw.toString();
```

Figure 10 An example using the JDom that PATL cannot handle.

Guided Normalization. The number and percentage of lines in the source clients that require guided normalization are shown in the last column of Table 3. As we can see from the table, a significant portion of lines involved in many-to-many rules, i.e., 65.0% of lines, requires guided normalization to correctly perform the transformation. This result indicates (1) guided normalization is necessary; (2) guided normalization is only performed on necessary lines but not all lines, avoiding excessive change to the layout of the source code.

Besides guided normalization, a conversion to three-address code is performed at the beginning of the transformation, and the introduced temporary variables will be inlined at the end of the transformation. In our experiment, all temporary variables are inlined at the end. This result indicates that the conversion to three-address code would not significantly affect the source code, either.

In particular, most lines requiring guided normalization exist in the clients of the Swing/SWT. This is because a lot of UI elements are first initialized and then added to their parents, such as the following example, in which we need to first swap the statements at line 2 and line 3 and then the rule in our running example becomes application.

```
button = new JButton();
button.setText("OK");
panel.add(button);
```

Unspecifiable Cases. As shown by the column "U" in Table 3, there are in total 84 lines of code whose transformation patterns cannot be captured by PATL rules. These unsupported mapping patterns are summarized into the following three categories.

The first category is that type mapping does not form a function. For example, a class may split into two classes, each inherits part of the functionalities of the original class. Since in the meta variable declaration we only allow to map one class to another class, we cannot write a rule for such cases. Most lines, 74 out of 84 belong to this category.

The second category is that the transformation needs high-level coordination between match instances, and only one such instance of three lines is observed. This instance is from Jdom/Dom4j. Below is the source client code (left of Figure 10), which converts elements in `rulesElements` into a string in `rulesToRegister`. An ideal transformation of this case is shown on right: instead of directly converting each element into string, they are written into a `StringWriter` to get the concatenated string. While we can write rules to generate the first three lines of the target code, We cannot generate the call to `sw.toString()` at the end of the target client code because there is no corresponding source statement. Generating this type of "closing" statement remains future work.

The last category relates to the use of `JPopupMenu` in Swing. In the evaluation 7 out of the 84 lines belong to this category. In Swing, to show and hide a pop menu, we need to implement a listener in which we write code to show and hide the pop menu. In SWT, we only need to call a method `setMenu`. To perform this transformation, we need to match both a method definition as well as the method body, which is not supported in PATL.

Warnings. As shown in the "W" column in Table 3, there are in total 30 lines of code on which warnings are generated. There are several reasons why warnings are generated. The first one, also the largest category, is that some code pieces cannot be transformed or shifted together because of dependencies among statements. 18 out of 30 lines belong to this category. The second one is that one line is matched by multiple "-" block in different rule instances. 2 out of 30 lines belong to this category. These two categories actually show the usefulness of guided normalization: unsafe transformations are disabled by guided normalization.

The last category is that the matched code is scattered in several methods. Since we do not move statements across method boundaries, our approach reports warnings on these lines. 10 out of 30 lines belong to this category. This shows a limitation of our current approach.

Untransformable code. As shown in the "I" column in Table 3, there are in total 50 lines of source code that are impossible to transform. This is because we cannot find any counterpart in the target API for a portion of the source API, and any lines invoking this portion cannot be transformed. The portion of untransformable API in the Google Calendar case is confirmed by the official Google Calendar upgrade guide, which states some functionalities are removed and will not be supported any more.

8 Related Work

Pattern finding in programs. Several works have been done in enhancing the pattern finding technique to enable finding program patterns with simpler rules [28, 7]. Among them, the most related is the approach proposed by Brunel et al. [7], which enables users to specify a sequence of consecutive statements to match different form statements against patterns with same semantics via model checking. The match part of our transformation technique is similar to this matching mechanism, while differently, besides matching, our language also performs transformation using these consecutive statement patterns.

Transformation languages. A lot of different program transformation approaches have been proposed for different purposes.

Some of these approaches [24, 18, 3, 36, 21, 20] are specialized for handling one-to-many mappings: one method invocation is replaced by a sequence of invocations. Among them the most related are Twinning [24] and SWIN [18], both of which are designed for handling the API migration problem with one-to-many mapping rules. Our basic semantics and the type mapping design are similar to them. Another related approach is update calculus [21, 20], which ensures the type-safe update of programs. Our approach currently does not ensure type safety, and potentially can be combined with update calculus and SWIN to ensure type safety. Overall, compared with these languages designed for one-to-many mappings, our approach supports many-to-many mappings, the importance of which is recognized by several empirical studies [4, 34] as well as our evaluation.

Besides, a number of program transformation languages or frameworks can be used to handle program transformation involved in many-to-many mappings.

SmPL [27] is a transformation language designed to document and automate the collateral evolutions of large C programs with patch rules. Our syntax for specifying the three patterns are inherited from semantic patches. SmPL also includes a state check [7] to check transformations that may potentially break data dependency relations, and a failure will be report on Case-2 in introduction (which our approach successfully handles). Stratego [6] is a general purpose program transformation language. It handles the context-sensitive

transformation with the support of dynamic rules: dataflow facts in can be collected and propagated [25] and users can write rules at places where context information is not yet available. TXL [9] is another general purpose transformation language, which allows acquiring the context information during abstract syntax tree traversal. Lacey and De Moor [17] propose a graph rewriting language where conditions on execution paths can be specified using temporary logic. Crossver [32] proposed a way to combine dataflow analysis with aspect-oriented programming to transform programs.

Overall, though these languages have provided more general and expressive operations for matching and transforming programs, they are relatively low-level in API transformation cases and require more effort to create correct and generic transformation. Differently, our language focuses on a more restrict but simpler language interface for the API transformation tasks.

Automated Transformation for APIs. Several approaches try to further reduce the cost of program adaptation between APIs by automatically discovering the transformation program. Typical approach includes recording the API refactorings and replay them on the client code [13, 12], and analyzing manually adapted code pieces [2, 23, 22] or clients [40, 39]. Currently, none of these approaches guarantee the safety of many-to-many transformations. Our approach can be potentially combined with those approaches to reduce the effort of writing code.

Normalization. Normalization techniques are commonly used in compiler optimization[15, 1, 14, 37, 8], semantics-preserving refactoring [26, 31] and procedure extraction [16].

Komondoor and Horowitz [16] presented a normalization technique that helps to move a set of identified statements to form a consecutive sequence, for program extraction purpose. However, due to the different domain requirements, their approach [16] handles only the situation that the normalization target is one sequence of statements, while our approach is required to normalize multiple sequences at the same time. As a result, we restrict the transformation primitives to retain the semantics-preserving property with the more complex normalization goal. Furthermore, transforming programs between APIs also requires performing renaming between aliases, which is not supported by Komondoor and Horowitz's approach [16] as it is not a requirement in code extraction.

Code motion [15] is an optimization technique which, when applied to if-branches, behaves similarly to our shift operation. However, these operations are not designed for user-specified program transformations. In addition, different from their approach and other compiler optimization approaches [1, 14, 37, 8], normalization in our approach are guided by match instances so that most original program structure can be kept after normalization.

9 Conclusion

In this paper we have presented PATL for safe transformation of complex programs between different APIs with simple Many-to-Many mapping rules. The key insight for PATL is to bridge the gap between simple in-block transformation semantics and complex program structures using guided normalization, which automatically changes the program so that code in different forms can be transformed in a unified manner. We applied PATL to three real world program transformation cases. The evaluation showed that PATL is expressive in handling real world scenarios, and with the help of guided normalization, only a small amount of manual resolution is required.

Acknowledgements. We thank Michael D. Ernst and the anonymous reviewers for their valuable comments and suggestions to improve the paper.

References

1 Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley, 2006.

2 Jesper Andersen, Anh Cuong Nguyen, David Lo, Julia L. Lawall, and Siau-Cheng Khoo. Semantic patch inference. In *ASE*, 2012.

3 Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *OOPSLA*, 2005.

4 Thiago Tonelli Bartolomei, Krzysztof Czarnecki, Ralf Lämmel, and Tijs van der Storm. Study of an api migration for two xml apis. In *SLE*, 2010.

5 G.M. Bierman, M.J. Parkinson, and A.M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report UCAM-CL-TR-563, University of Cambridge, Computer Laboratory, 2003. URL: `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-563.pdf`.

6 Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2), June 2008. URL: `http://dx.doi.org/10.1016/j.scico.2007.11.003`, `doi:10.1016/j.scico.2007.11.003`.

7 Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching: using temporal logic and model checking. In *POPL*, 2009.

8 David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *CC*, 1986.

9 James R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, 61(3), 2006.

10 Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

11 Danny Dig and Ralph Johnson. How do APIs evolve? a story of refactoring: Research articles. *J. Softw. Maint. Evol.*, 18(2), 2006.

12 Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph E. Johnson. *ReBA*: *re*factoring-aware *b*inary *a*daptation of evolving libraries. In *ICSE*, 2008.

13 Johannes Henkel and Amer Diwan. Catchup!: Capturing and replaying refactorings to support api evolution. In *ICSE*, 2005.

14 Aditya Kanade, Amitabha Sanyal, and Uday P. Khedker. A PVS based framework for validating compiler optimizations. In *SEFM*, 2006.

15 Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *PLDI*, 1992.

16 Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. In *POPL*, 2000.

17 David Lacey and Oege de Moor. Imperative program transformation by rewriting. In *CC*, pages 52–68, 2001.

18 Jun Li, Chenglong Wang, Yingfei Xiong, and Zhenjiang Hu. Swin: Towards type-safe java program adaptation between apis. In *PEPM*, 2015.

19 Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. How does web service api evolution affect clients? In *ICWS*, 2013.

20 Deling Ren Martin Erwig. Type-safe update programming. In *ESOP*, 2003.

21 Deling Ren Martin Erwig. An update calculus for expressing type-safe program updates. *Science of Computer Programming*, 67, 2007.

22 Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: Generating program transformations from an example. In *PLDI*, 2011.

23 Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to API usage adaptation. In *OOPSLA*, 2010.

24 Marius Nita and David Notkin. Using twinning to adapt programs to alternative APIs. In *ICSE*, 2010.

25 Karina Olmos and Eelco Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In *CC*, 2005.

26 William F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

27 Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *Eurosys*, 2008.

28 Nicolas Palix, Jean-Rémy Falleri, and Julia Lawall. Improving pattern tracking with a language-aware tree differencing algorithm. In *SANER*, 2015.

29 Benjamin C. Pierce. *Types and Programming Languages*. 2002.

30 Mark Pilgrim. *Dive Into Python 3*. Apress, 2009.

31 Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

32 Kouhei Sakurai and Hidehiko Masuhara. Crossver: a code transformation language for crosscutting changes. In *AOAsia/Pacific Workshop*, 2014.

33 Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and extensible renaming for java. In *OOPSLA*, 2008.

34 Thiago Tonelli Bartolomei, Krzysztof Czarnecki, and Ralf Lämmel. Swing to SWT and back: Patterns for API migration by wrapping. In *ICSM*, 2010.

35 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, 1999.

36 Louis Wasserman. Scalable, example-based refactorings with refaster. In *WRT*, 2013.

37 Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2), April 1991.

38 Daniel Weise and Roger Crew. Programmable syntax macros. In *PLDI*, 1993.

39 Qian Wu, Guangtai Liang, Qianxiang Wang, and Hong Mei. Mining effective temporal specifications from heterogeneous API data. *J. Comput. Sci. Technol.*, 26(6), 2011.

40 Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining API mapping for language migration. In *ICSE*, 2010.

Towards Ontology-Based Program Analysis*

Yue Zhao[1], Guoyang Chen[2], Chunhua Liao[3], and Xipeng Shen[4]

1 **Department of Computer Science, North Carolina State University, USA**
yzhao30@ncsu.edu
2 **Department of Computer Science, North Carolina State University, USA**
gychen1991@gmail.com
3 **Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, USA**
liao6@llnl.gov
4 **Department of Computer Science, North Carolina State University, USA**
xshen5@ncsu.edu

Abstract

Program analysis is fundamental for program optimizations, debugging, and many other tasks. But developing program analyses has been a challenging and error-prone process for general users. Declarative program analysis has shown the promise to dramatically improve the productivity in the development of program analyses. Current declarative program analysis is however subject to some major limitations in supporting cooperations among analysis tools, guiding program optimizations, and often requires much effort for repeated program preprocessing.

In this work, we advocate the integration of ontology into declarative program analysis. As a way to standardize the definitions of concepts in a domain and the representation of the knowledge in the domain, ontology offers a promising way to address the limitations of current declarative program analysis. We develop a prototype framework named PATO for conducting program analysis upon ontology-based program representation. Experiments on six program analyses confirm the potential of ontology for complementing existing declarative program analysis. It supports multiple analyses without separate program preprocessing, promotes cooperative Liveness analysis between two compilers, and effectively guides a data placement optimization for Graphic Processing Units (GPU).

1998 ACM Subject Classification D.3.4 [Programming Languages] Processors—Compilers

Keywords and phrases ontology, compiler, program analysis

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.26

1 Introduction

Program analysis [45] is a common way for deriving various properties of a program from its code. It is fundamental for many aspects of modern computing, including program optimizations, vectorization and parallelization, performance or correctness bug identification, task scheduling, and so on.

* This material is based upon work supported by DOE Early Career Award (DE-SC0013700), and the National Science Foundation (NSF) under Grants No. 1455404, 1455733 (CAREER), and 1525609. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE and NSF.

There are mainly two ways to implement a program analysis. A traditional way is imperative, in which, thousands of lines of code (often in some imperative programming languages) is developed based on some compiler framework for analyzing program constructs, types, control or data flows to infer certain properties of the target program. Programmers typically need to go through some steep learning curve about the structure and internal details of a complex compiler, while the results are often unsatisfactory: The code is often difficult to maintain, and bugs are common [63]. The analysis, being specific to a particular compiler, is hard to extend, to compose, or to reuse for other compilers.

The second approach, *declarative program analysis*, has been proposed to overcome the productivity issues [56, 32, 17]. With it, the developers just need to define some abstract domains and then use some logic programming language (e.g., Datalog [60]) to describe the analysis rules that govern the relations or properties of interest. Some automatic tools can then automatically do the inferences over a certain representation of some relations in the target program to find out the wanted relations or properties of the program. Experiments have shown that, with this approach, the code size of a program analysis often reduces by orders of magnitude compared to the imperative approach, and the analyses become easier to maintain and extend [13]. Moreover, with the substantial improvement in optimizations of the logic processing engines (e.g., bddbddb [59] and numerous optimizations to inference engines [62]), the performance and scalability concerns of declarative program analyses have been largely resolved.

In this work, we aim to further improve this promising paradigm of program analysis, particularly, to investigate solutions to three most important limitations of the current declarative program analysis:

- *Cooperations.* By expressing the analysis at a high level, different analysis tools could potentially reuse an analysis, and different analyses could get composed together into a more sophisticated analysis. However, in practice, these benefits have been difficult to achieve in general, due to the differences in the analysis-specific representations of programs and relations. In one analysis, the domain may be variable names and heap addresses, and the relation may be "assigning one address to a variable"; in another analysis, the domain may be expressions and the relation may be "calculated before a program point". To compose the two analyses, the variable names and heap addresses in the first analysis may have to be mapped to the expressions in the second domain, which would require much code development (likely entwined with the code in the compilers), especially if the two analyses were developed by different users based on different compilers that use different intermediate representations (IRs).
- *Optimizations.* So far, explorations of declarative program analysis have been focused on understanding program behaviors (largely for the purpose of debugging), for which, program-level knowledge has been enough. It is, however, insufficient for another important purpose of program analysis, guiding program optimizations. For the multi-facet dependence of performance, program optimizations often need knowledge from various sources: the program itself, the hardware, the algorithms, the program input datasets, and various domain-specific or problem-specific knowledge. Consequently, to provide useful optimization guidance, declarative program analysis must support the representations of the various kinds of knowledge, and allow easy linkage among them, even if the various kinds of knowledge may come from different sources. The analysis-specific nature of the current declarative analysis designs offers poor support to these needs.
- *Preprocessing.* A declarative program analysis typically requires some preprocessing to extract useful relations from the target programs to build up a relational database. This

step hurts the productivity benefits of this approach: As it is usually tightly coupled with some compiler framework, it is tedious and error-prone to develop. What makes this especially problematic is that different program analyses often *use different relations or ways to define the same or similar relations*. As a result, preprocessing needs to be developed for almost every newly developed program analysis, seriously throttling the productivity benefits of declarative program analysis.

In this work, we advocate the integration of ontology into declarative program analysis to address the three limitations all together. Our proposal comes from the observation that all the three major limitations essentially stem from a single fundamental shortcoming in current declarative program analysis: the lack of a systematic conceptual framework to govern the definition, representation, and organization of the various kinds of knowledge (relations in a program, rules, domains, hardware configurations, etc.) related with program analysis. The ad-hoc analysis-specific approach used in today's designs of declarative program analysis is the fundamental reason for the much effort required for preprocessing, and the barriers for supporting cooperations and optimizations.

Ontology, a concept originated from Philosophy, refers to the study of the nature of being, as well as the basic categories of being and their relations [47]. In recent decades, it has become a branch in Information Science, serving as the primary way to standardize the concept definitions and knowledge representations for a domain. It includes three concrete components: A standard vocabulary and definitions of some common concepts and their relations in a given domain, a standard format (e.g., the Web ontology Language (OWL)) for representing the various instances and concepts in any concrete problems in the domain and their relations, and a whole set of *tools* that have been developed in the last several decades for the development of an ontology and automatic logic inference upon it.

The key idea in our proposal is to leverage ontology to help standardize the definitions of domains, relations, and other concepts in program analysis, and to establish a single flexible representation of program constructs as well as other kinds of knowledge related with program analysis and its usage. With that, knowledge from various sources may be linked seamlessly as long as they follow the standardized representation. Sharing the same conceptual framework and set of terminology, different program analyses will be easy to compose and interoperate together. The standardization will also make it possible to develop a single comprehensive database of the relations of a program to serve for various program analyses, removing the needs for the separate development of preprocessing for each analysis.

Ultimately, it would be desirable to establish a standard Program Analysis ontology to describe programs, analysis, and related concepts—liken how the Semantic Sensor Network ontology by W3C [15] facilitates the work in the sensor network domains. Reaching that goal would require the coordinated effort from the community and goes beyond the scope of this paper.

This work instead focuses on the following four-fold objectives:

- To introduce the idea of integrating ontology into program analysis, and explain the concept of *ontology-based program analysis* and its potential benefits.
- To investigate the feasibility of having a single representation of a program in ontology to facilitate various program analyses and hence reduce the much effort for developing program preprocessing as required in current declarative program analysis.
- To validate the promise of ontology as the representation of various kinds of knowledge related to program optimizations, and hence extend existing declarative program analysis to guide program optimizations.

- To confirm the benefits of ontology for facilitating easy cooperations of different analysis tools.

To reach the four objectives, we have developed a prototype framework named PATO (which stands for Program Analysis Through ontology). In this prototype, we explore the use of several principles to define a concept-proof ontology for C program representations. Based on it, we have developed five program analyses: canonical loop analysis, pointer analysis, control flow graph construction, data access pattern analysis, and GPU data placement guidance. These analyses differ in domains, relations, scopes, and intended usage. Our experiments show that a single ontology-based representation can successfully support all these analyses (without separate preprocessing per analysis). The analyses inherit the productivity benefits of declarative program analysis, reducing the lines of code by tens of times compared to imperative implementations. Using the liveness analysis on two compilers (ROSE [4] and LLVM [39]), we confirm the benefits of ontology for promoting cooperations among different analysis tools. And using GPU data placement optimization, we demonstrate the seamless linkage of various sources of knowledge (programs, domain experts, and hardware) enabled by ontology, and reveal the potential of ontology-based program analysis for guiding program optimizations.

We acknowledge that this work is just the first step towards ontology-based program analysis. The ontology implemented in this work is for only the representations of C programs (and GPU data placement), and the analysis has not exploited the potential of ontology much beyond logic programming upon ontology-based program representations. A full development of ontology-based program analysis would require also the ontology and formal definitions of common concepts in program analysis (e.g., dominator, post order, alias), their relations, and some deep investigations of the opportunities that ontology may bring to program analysis. Through this first step, by demonstrating the promise of ontology-based program analysis, we hope that this work will stimulate further studies by the community into this promising direction.

2 Background

Ontology is a concept originating in Philosophy, referring to the study of the nature of being, as well as the basic categories of being and their relations [47]. In recent decades, it has become a branch in Information Science for representing the knowledge in a particular domain. In this context, an ontology is a formal explicit description of a domain's knowledge, including concepts (or classes), properties of each concept (or relations) and individuals (or instances of classes) [53]. An ontology is often visualized as a graph in which nodes indicate concepts and edges indicate relations between concepts. It is often represented in some standard triple format as we will describe later in this section. Developing an ontology for a domain has many benefits, including 1) making domain knowledge explicit to expose what is known and what is unknown, 2) enabling knowledge interoperability by providing a common taxonomy and vocabulary, 3) providing knowledge reuse since the ontology is a persistent knowledge base, and 4) facilitating knowledge validation and reasoning using existing inference engines (or reasoners).

Figure 1 shows an example borrowed from an introductory article of ontology [47]. A class (e.g., "Winery") describes concepts in a domain. A class can have subclasses, representing the "kind of" relations among concepts (e.g., "red wine" may be a subclass of "wine"). A class can have many individual instances (e.g., "Chateau Lafite Rothschild Pauillac" is an instance of the class "Pauillac"). A property describes the attribute of a class or

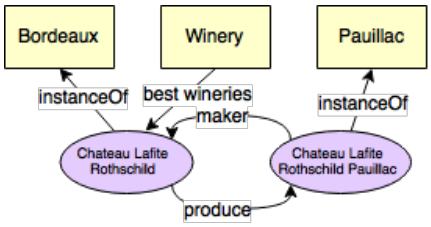


Figure 1 An example ontology for the domain of winery.

instance. It is usually represented as an edge between classes or instances. For instance, the "maker" property in Figure 1 shows that the maker of "Chateau Lafite Rothschild Pauillac" is "Chateau Lafite Rothschild". A property may have some constraints, which describe the value type, allowed values, the number of the values (cardinality), and other features of the values which the property can take.

The theory foundation of ontology is Description Logic (DL) [36], a family of formal knowledge representation languages for formal reasoning on the concepts of a domain. DL is expressive enough to build sophisticated knowledge bases while still supporting efficient inference. It has a popular standardized dialect, Web ontology Language or OWL [44]. DL allows the use of *axioms* to describe a knowledge base. For instance, an axiom $\mathsf{Person(Alex)}$ describes that "Alex" is an instance of class "Person", and $\mathsf{Person} \sqsupseteq \mathsf{Human}$ describes the subsumption relationship between concept Person and concept Human. DL languages could use different grammars. Conventionally, a simple yet uniform format to represent axioms is $(\mathsf{subject}, \mathsf{property}, \mathsf{object})$ triple. Axioms in DL languages can all be mapped to such a format. For instance, in $\mathsf{Person(Alex)}$, "Alex" is the subject, "instance of" is the property, and "Person" is the object. Visually, it is an "instance of" edge flowing from the "Alex" node to the "Person" node in an ontology graph.

Through decades of development, a large body of tools (e.g., Stanford Protégé [25], SWI-Prolog Semantic Web Library [62], etc.) have been developed for creating ontologies and automatic reasoning upon an ontology-based knowledge base, which enables automatic questions-and-answers, consistency check of the knowledge base, derivation of new knowledge, and so on.

Like declarative program analysis, most of these tools leverage logic programming languages (e.g., Prolog) for inferences. In logical programming, a program is composed of a list of rules written in the form of *clauses*:

$$\mathsf{H} \text{:-} \mathsf{B}_1, \ldots, \mathsf{B}_n.$$

which reads as

$$\mathsf{H\ is\ true\ if\ B}_1\ \mathsf{is\ true\ and\ ...\ B}_n\ \mathsf{is\ true}.$$

H is called the *head* of the rule and $\mathsf{B}_1, \ldots, \mathsf{B}_n$ is called the body. When the body is empty, the rule becomes *fact*; for instance, $\mathsf{variable(a)}$ states the fact that a is a variable.

3 Overview

By offering a generic way to represent the knowledge in a domain, ontology simplifies the accumulation and share of various kinds of knowledge among people or software agents. At the same time, it allows automatic analysis and utilization of knowledge through high-level

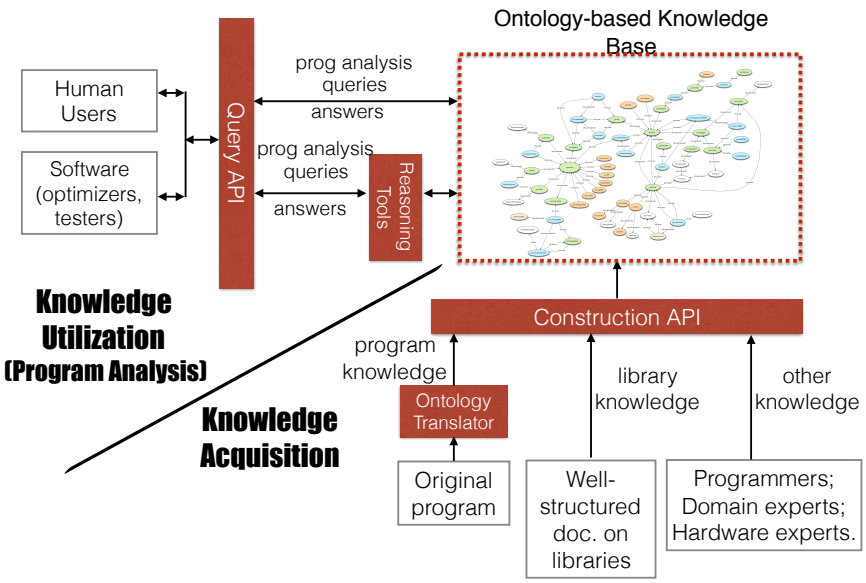


Figure 2 Overview of using ontology for program analysis.

declarative logic programming, thanks to its description logic foundation. These properties make it potentially valuable for facilitating program analysis which is essentially about reasoning about the knowledge related to a program.

Figure 2 illustrates the basic idea of ontology-based program analysis. It centers around a knowledge base built upon ontology. The knowledge base may consist of the basic knowledge about the code of the target program, as well as other knowledge (e.g., architecture attributes) relevant to the program analysis. An *ontology converter*, equipped with a parser, derives the basic program knowledge from the code of the program, expresses the knowledge in a standard format, and puts it into the knowledge base. This basic knowledge may include the structures and components (control blocks, data structures, etc.) of the program. In addition, the knowledge base may include some knowledge that could be imported about some libraries, or directly input by a domain expert about some properties of the program or the hardware it executes on (for optimizations).

Built upon description logic, ontology-based program analysis keeps the conveniences of declarative program analysis. Rather than writing thousands of lines of code inside a complex compiler, users can simply write some logic queries about the kind of properties (e.g., which loops are canonical loops) of the program that they want to know. These queries should follow some ontology query APIs. The APIs will then return the answers that are automatically obtained from the ontology-based knowledge base. For complex queries, it can leverage many existing ontology reasoning tools [62, 55]. Users of the ontology-based knowledge base can be humans or software agents (e.g., tools for program optimizations or testing.)

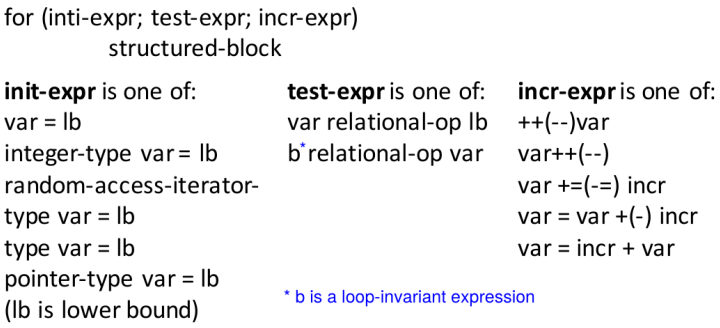


Figure 3 Specification of a canonical loop (derived from OpenMP manual [48]).

Example. We use canonical loop analysis to illustrate how the idea of ontology-based program analysis works. A canonical loop is a type of well-structured loop conforming to specifications as shown in Figure 3. Because of its regular structure, it has been the focus of many studies on parallelization and loop optimizations [35, 41, 16].

The goal of *canonical loop analysis* is to recognize whether a loop is in a canonical form. Traditional implementations of the analysis (e.g., the implementation in the ROSE compiler [4]) contains hundreds of lines of code for examining the IR of a loop. The code is tied to a particular internal data structure of the chosen compiler, hard to port to another compiler or maintain.

In a traditional program analysis development, the task requires an insertion of a separate pass over some intermediate representation of the whole program, which may need the development of thousands of lines of code. For example, in the ROSE compiler [4] (a source-to-source compiler broadly used in High Performance Computing), the pass works on an Abstract Syntax Tree (AST). To analyze the code at that level, a programmer needs to implement many lines of code written in procedural languages (C/C++). The canonical loop analysis in the ROSE compiler consists of 380 lines of source code for examining the representations of the structure of each loop and check them against the conditions in Figure 3.

In ontology-based program analysis, the process is simpler. The programmer needs to invoke some provided ontology converter on the code of the target program. An ontology-based knowledge base is then produced to capture the program constructs, components, and their relations. The programmer then just needs to use a declarative logic programming language to describe rules governing the forms that a canonical loop should conform. Treating those rules as queries on the ontology of the program, existing logic reasoners can then automatically find all the canonical loops in the target program. For example, a fragment of C code is shown in Listing 1. The code's corresponding ontology representation is shown in Listing 2. Each line is a triple: (subject, predicate, object). The numbers in the subjects or objects are the line and column numbers of the beginning and ending positions of a language construct in the source code. (Next section explains the triples in the example in details.) The different program constructs can be easily extracted by Prolog queries like those in Listing 3.

Allowing the use of logic programming, ontology-based program analysis inherits the productivity benefits of declarative program analysis. More importantly, it overcomes the three aforementioned shortcomings of existing declarative program analysis by leveraging ontology for standardizing the concept definitions in a domain and the flexible representation of various sources of knowledge.

Listing 1 Example C code fragment.

```
0 // s.c
1 int a = 0;
2 int foo() {
3  for (int i = 0; i < 10; i++) {
4   a = a + i;
5  }
6  return 0;
7 }
```

Listing 2 Sample ontology for C snippet ("x rdf:type y" indicates that y is the type of the code segment in the range x).

```
1 ('3:1,5:1', rdf:type, 'ForStatement')
2 ('3:6,3:14', rdf:type, 'VariableDecl')
3 ('3:6,3:10', rdf:type, Variable')
4 ('3:1,5:1', 'hasForInit', '3:6,3:14')
5 ('3:1,5:1', 'hasForTest', '3:17,3:22')
6 ('3:1,5:1', 'hasForIncr', '3:25,3:27')
7 ('3:1,5:1', 'hasBody', '3:30,5:1')
```

4 Challenges and Solutions

The challenges for integrating ontology into program analysis exist in each of the four main steps: the design of an ontology for the domain, the generation of the knowledge, the utilization of the knowledge base, and the design of the entire framework. In this section, we discuss each of the challenges and present some principles we use to address them. At the end, we describe PATO, the prototype framework we have developed to do program analysis upon ontology-based program representations.

4.1 Ontology Design

Challenges. To create an ontology for any domain, the first primary task is to define the vocabulary to be used in the domain. That includes the definition of the concepts, properties, and restrictions in the domain. These definitions establish the conceptual terms and their relations of the ontology-based knowledge base for the domain. Even though there are some de facto procedures on designing an ontology [47], program analysis has some special challenges. Program analysis has a large variety of tasks (e.g., loop analysis, data access pattern analysis, alias analysis, dependence analysis, liveness analysis, busy expression analysis, etc.) involving a huge set of diverse concepts and relations. Even a larger variety exists in the input programs. So the first question to use ontology for program analysis is how to design an intuitive, efficient and flexible ontology that can facilitate the various program analyses and input programs.

Solutions. In this work, we focus on the design of ontology for representing programs of a particular programming language (C). The design of the complete ontology of program analysis is left to future work.

Through our explorations, we have found the following three principles helpful.

Listing 3 Sample analysis rules.

```
1 isForStatement(Loop) :-
2  rdf(Loop, rdf:type, c:ForStatement).
3 hasForInit(Loop, InitExpr) :-
4  rdf(Loop, c:hasForInit, InitExpr).
```

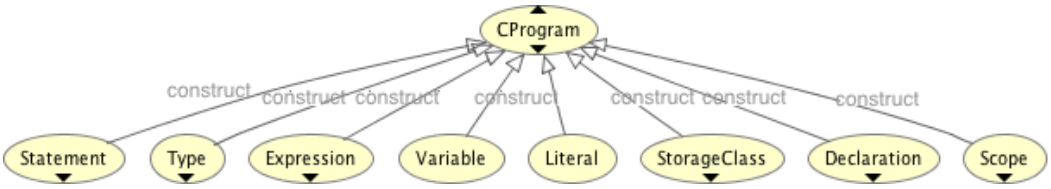


Figure 4 Top-level ontology for C program analysis.

Language standard-oriented design. When designing the vocabulary of an ontology, it helps if one starts with reusing language constructs and their categorizations defined in the standard of the programming language of the target programs. Despite the variety of the input programs, they are all artifacts following the particular programming language. With the constructs and their categorizations of the programming language covered, we can easily express programs using the language in the ontology-based knowledge base. This approach helps achieve a good coverage of the input programs with a vocabulary familiar to users. For example, to model C programs in an ontology, we followed the standard of C99 and enumerate all program constructs and concepts in a top-down fashion. Figure 4 shows a fraction of the top-level ontology for C programs. The main concepts in the domain include variables, expressions, statements, and so on. The "construct" on the edges indicate these vocabularies are the basic constructs in the C program domain.

Being generic in property designs. Besides classes of concepts, an ontology also contains a vocabulary for properties (or called relations). For example, an Expression instance may have some Type, an Identifier may refer to some Definition. Because there may be many properties to express, in our practice, we follow a principle trying to define properties in a generic way, and encode semantic meanings into concepts whenever possible. That allows the possible use of a small set of properties and their combinations to express a large number of possible properties in a program. For example, when describing *an identifier has static storage class*, one approach is to define a property hasStaticStorage and use it like (someVar hasStaticStorage true). Alternatively, we may define the concept Storage as a class and use a simpler property hasStorage as (someVar hasStorage static), where static is a member of Storage. There are several benefits for this second choice. First, using generic properties makes the set of property vocabularies small and thus easy to manage. For example, hasStorage is used to describe all storage classes instead of creating specific properties for each storage class. Second, stripping semantics from property make writing logic rules more flexible. For example, the knowledge of (someVar hasStorage static) can be queried by the keyword hasStorage. Otherwise, users may need to use many specific keywords as hasStaticStorage, hasExternStorage, and so on.

Continuous enrichment. In our exploration, we find that continuous enrichment of the ontology vocabulary can be helpful. For instance, in a canonical loop analysis, a user gives the description of the concept of a canonical loop. If that concept turns out to be needed frequently (by many users), the concept could then be integrated into the ontology

framework to save the need for repeated descriptions. Given that the program analysis ontology is intended to be used by a community, a complexity is that different users may use different names for a single user-defined relation (e.g., canonical loop), making the detection of the repeated use of a user-level concept difficult. Ontology-based logic reasoners come in handy. As the descriptions of user-defined relations all use the vocabulary in the same ontology, the reasoners can easily do a logic reasoning on the descriptions to decide whether two user-defined relations are equivalent. After recognizing the frequently needed user-level concepts, such concepts can be added into the standard ontology for future reuse.

Based on the three principles, we came up with an ontology for C program representations to facilitate C program analysis. It contains 178 concepts and 68 properties. It is not intended to be complete (e.g., many program analysis concepts are not yet defined), but is sufficient for examining the support of a single ontology to multiple different program analyses as we will show later in this paper.

4.2 Knowledge Generation

Challenges. Building up a knowledge base is essential for any application of ontology. For program analysis, the knowledge base shall include the important knowledge related with the to-be-analyzed program. There are three main questions to answer. *(1) ontology converter.* How to construct an ontology converter that can automatically convert a given program into the ontology representation needed by many common program analyses. *(2) Naming.* A program may contain functions, statements, expressions, variables, and so on. Any of them may appear many times in different locations. A challenge is what naming scheme the ontology should use to reference each reference to avoid ambiguity. *(3) Mapping.* One of the objectives of ontology-based program analysis is to facilitate the cooperations among different compilers and other program analysis tools. A difficulty is that they may have different internal representations of a program. To make them able to interact based on the program ontology, the instances of program constructs in the knowledge base should be possible to map to some common ground meaningful to the different tools. One intuitive choice of such common ground would be the source code of the program. That will also make it easy for human to collaborate with program analysis tools. A complexity in using source code for references is how to make the naming robust to code changes. *(4) Space.* The knowledge about a program can be tremendous. Besides the basic knowledge directly driven from the program, there could be many other kinds of higher-level knowledge such as canonical loops, data dependences among statements, and so on. The higher-level knowledge is derivable from the lower-level knowledge. The derivation through reasoning may take non-trivial time. But if all this knowledge is saved in the knowledge base, the space cost could be large. How to strike a good balance is important for practical usage of this new program analysis paradigm.

Solutions. We come up with the following solutions to these challenges.

Ontology converter. Our experience shows that an ontology converter can be easily created through a translator built on top of a source-to-source compiler such as ROSE [4]. A source-to-source compiler usually produces an abstract syntax tree (AST) representation which is close to the input code. The translator traverses the AST of the program to get structural and semantic information, which is then stored into the knowledge base as ontology. The program constructs are represented as individuals (i.e. instances) of some of the classes defined in the language ontology. Relations between them are represented by properties.

Naming and Mapping. In our naming scheme, we borrow the *internationalized resource identifiers (IRIs)* [28] that OWL uses. It helps avoid name conflicts. For the ontology of C program language, names of concepts are built directly from the corresponding terms used in the C language standard[1]. For example, the concept type is referenced as c:Type. An example IRI for the concept of types in a C program domain can look like "http://example.com/owl/CProgram:Type". Some aliases can be defined as short names for the prefix strings of a domain.

More care needs to be taken for designing the naming scheme for representing the instances of a program construct. Named constructs such as types, variables, functions can use the C++ qualified name concept to uniquely identify them. For an unnamed construct (e.g., an assignment statement) or a reference to named construct (e.g., variable reference), a common intuitive approach is to use its location in the source code of the program, such as file url, start location, end location where the location is a pair of (line number, column number). For instance, "http://my.com/file1.c, 3:1, 5:1" could refer to a loop that spans from the beginning of the third line to the beginning of the fifth line of file1.c. The problem with this scheme is that some minor changes to the original program may invalidate all the names of the constructs after the modification point.

We find *scoped IRI* useful to restrict the impact of a code change to the names. In *scoped IRI*, a name is composed of some qualified names and some relative locations in the source code. For constructs like functions, structures, global variables, we add their scopes before their names. Other constructs within these constructs are named by their locations, while the line numbers are relative to the start line number of their surrounding constructs rather than the beginning of the source-code file. Using this method, a global variable declared in the first line of a file "s.c" (from column 5 to 6) can be named as s.c::1:5,1:6, while a variable declared on the first line inside a function foo in file "s.c" (from column 6 to 10) can be named as s.c::foo()::1:6,1:10. Thus, if there is some change of the code, only the names in the same scope as the changing point is need to be updated.

Space. To address the space challenge of putting everything into a knowledge base, we split the knowledge base into a core knowledge base and multiple loadable supplemental knowledge bases. The core knowledge base is always loaded and others are loaded as needed. We also design a cache-like management mechanism to alleviate the problem. It maintains a buffer to store derived supplemental knowledge. When the upper limit of the buffer gets reached, it starts to evict some of the stored knowledge (which would need to be rederived when needed). For the eviction policy, there can be multiple choices: least-recently-used (LRU), least-frequently-used (LFU), and their variants.

4.3 Knowledge Utilization

Challenges. Some challenges also exist in the utilization of the knowledge base for program analysis. *(1) Efficiency.* In many cases (e.g., analyzing a large program), the runtime efficiency of conducting a program analysis could be important. A question to answer is whether the improved productivity of the new paradigm hurts the runtime efficiency and if so, how to improve the efficiency. *(2) Generality.* Using declarative programming languages could be awkward for some usage cases, especially when they involve some mathematical computations. Such cases however do exist in some program analysis and optimizations, for instance, when they relate with some performance models (an example is the data placement

[1] We follow the naming convention of UpperCamelCase for classes and lowerCamelCase for properties.

optimizations Section 5 will describe). Effectively overcoming such limitations is important for the general applicability of ontology-based program analysis.

Solutions. We address these issues by both creating some shortcuts and leveraging the features of existing ontology tools.

Efficiency. Recent years have seen some significant improvement of the performance of logic reasoners [55, 9]. Many optimizations have been developed. For instance, in SWI-Prolog, the ontology is stored as relation triples of (subject property object) with C extensions and some indices are built for each element in the triples. So a search of a particular element can be done in constant time. Additional optimizations can be applied to queries. For instance, Prolog provides the *cut* operator (i.e., the ! symbol) to avoid unwanted backtracking in search. We find that following some existing guidelines when writing queries [12] can be quite helpful for quickly narrowing down the reasoner's search space.

In scenarios where the relevant knowledge base is simple and consists of straightforward facts in triples (e.g., some memory configurations), one may construct a customized lightweight parser in high performance languages, which can further help achieve good performance than going through a heavy-weight logic reasoner.

Generality. Ontology-based logic reasoning meets the needs of many typical program analyses, but is not quite suitable for expressing analyses that involve a lot of mathematical computations (e.g., a regression-based performance modeling). We find that a mechanism called *computable* [54] originating in Robotics can help resolve the issue. Computable is some special ontology entity that can attach procedures to some classes or properties. When the deduction rule queries the individuals of one of the classes or the properties, the associated procedures are invoked to compute individuals for the target class or property. The procedure can be written in a wide range of programming languages (e.g., C, C++, Python).

Besides allowing the direct input of queries by users written in logic programming languages, ontology-based program analysis also allows queries coming from a third-party software (which could be written in even imperative languages like C, C++ or Java.) For such cases, the ontology can be processed with libraries for those languages (e.g., the OWL API [31] or Prolog interface to foreign languages[7]). That offers the conveniences for existing software tools (e.g., a compiler) to easily leverage ontology for program analysis.

4.4 Framework Design and PATO

The final challenge is how to organize the various components together into a unified framework for program analysis. It includes choosing the DL language and the reasoner that suit the needs of program analysis, designing and implementing the APIs for both knowledge base construction and queries, and integrating all the components together into a complete cohesive framework that offers effective support to various program analyses.

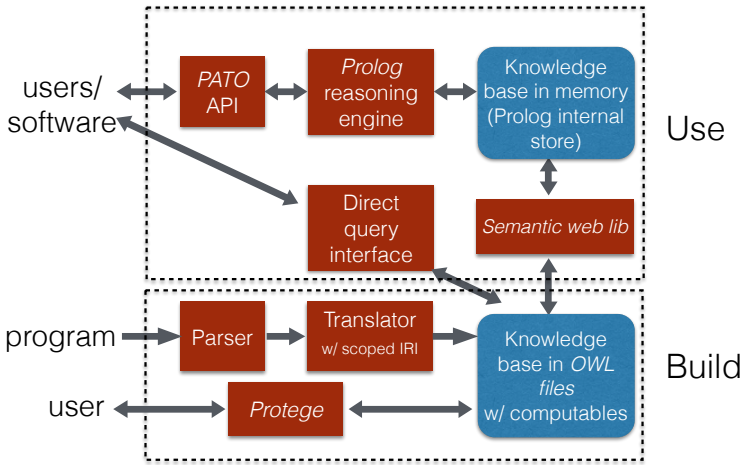


Figure 5 Structure of PATO.

To answer these questions, we have developed a prototype framework named PATO, which integrates all the aforementioned solutions to the various challenges, and leverages the power of existing ontology tools.

Figure 5 outlines the main components of PATO. The knowledge base in PATO can be in two forms, represented by the two round-corner boxes on the right part of the figure. On the disk, the knowledge is stored as OWL files. Each entry in the files is in an OWL triple (subject, property, object). For instance, (var1, hasValue, 0) means a variable has a constant value of 0. The reason for selecting this form is that OWL triple is one of the standard (and space-efficient) formats for ontology representations and is accepted by many ontology tools. Another part of the knowledge base is the *computables*, which are attached to the concepts and properties in the triple collection. The cache-like buffering mechanism is used for the space efficiency of the knowledge base.

Two of the primary ways to add knowledge into the knowledge base are shown at the bottom of Figure 5. In the first way, there is a parser (based on ROSE [4]) for converting an input program code into an AST preserving source level information. We have also developed a translator that translates the basic program knowledge on the AST into OWL triples and stores them into the knowledge base. During the translation, the translator uses the scoped IRI as the naming scheme. In the second way, we adopt Stanford Protégé [25], an interactive tool for ontology creation and manipulation. Through its GUI, a user can intuitively add entries into the knowledge base. The plugins of Protégé also provide many other features to the users, such as visualization, validation, and querying.

There are two ways to use the knowledge base. The first is shown at the top of Figure 5. It is through the SWI-Prolog reasoning engine. To use the knowledge base in this way, at the beginning, the OWL files are loaded into memory; through it, the OWl files are converted into an internal knowledge base through the existing Semantic Web Library [62]. The in-memory organization of the knowledge base features some efficient indexing schemes. When seeing a query from a user or software agent, the Prolog engine would start working on the internal knowledge base to provide the answers. Prolog provides a general interface for input logic rules and queries. We have further developed a higher-level set of API tailored to represent some common terms used in program analysis tasks (e.g., loops, functions, etc.), through them, the users may write even more concise descriptions.

The second way to use the knowledge base is shown as the shortcut (the diagonal path) in Figure 5. Through a direct query interface we have developed in C++, the user or

software agent may directly work on the OWL file collection (and computables) without going through Prolog. This shortcut is useful when Prolog is not available to a user or too costly to use in some special scenarios. For tasks which only need some simple fact search (without the need for much reasoning), this approach is more efficient than the Prolog-based approach because it avoids the overhead in Prolog interpretation and other associated cost. Both the Prolog and the shortcut approach can insert new knowledge (e.g., derived in a previous program analysis) into the knowledge base.

It is worth noting that the rich set of available tools on ontology proves helpful in our development and usage of PATO. Besides the aforementioned usage of Protégé [25] for ontology development, we find the FaCT++ reasoner [55] helpful for checking the consistency of the ontologies, the Prology *semweb* library [61] useful for loading, parsing, and manipulating ontology-based knowledge bases, and the Prolog engine [62] a convenient tool for inferences upon the knowledge bases.

5 Experience

This section describes our experience of using PATO for program analysis. To examine the feasibility of using a single ontology to support multiple different program analyses efficiently, we implement five types of program analysis on PATO: canonical loop analysis, pointer analysis, control flow graph construction, data access pattern analysis, and GPU data placement guidance. They differ in domains, relations, scopes, and intended uses. Using GPU data placement optimization, we examine whether ontology can indeed enable seamless linkage of various sources of knowledge (programs, domain experts, and hardware) and whether ontology-based program analysis can actually help guide program optimizations. Using the liveness analysis on two compilers (ROSE [4] and LLVM Clang [39]), we examine the benefits of ontology for promoting cooperations among different analysis tools.

For the interest of space, our description concentrates on the canonical loop analysis, the pointer analysis, and the cooperative liveness analysis. We briefly cover the other analyses and the GPU data placement experiment at the end. Without noting otherwise, each reported timing result is the average of 10 times of repeated measurements collected on a machine equipped with Intel Core-i5 CPU of 3.2GHz (8GB DRAM, 500Gb HDD hard drive) running Ubuntu 14.04. For the results that show large variances, we also report the statistics on the variances. The Prolog used is SWI Prolog, and the primary compiler is the ROSE compiler (EDG 4x-Based versoin) [4].

5.1 Canonical Loop Analysis

We first explain the Prolog code for canonical loop analysis. As mentioned earlier, canonical loop analysis (CLA) checks whether a loop is in a predefined canonical form. Our experiment uses the *canonical loop form* defined in the OpenMP specification [48], shown in Figure 3. The specification of an OpenMP canonical loop can be written as declarative Prolog rules as shown in Listing 4. We use italic font to distinguish variable from normal symbols.

In the Prolog specification, the *head* cannonicalLoop(Loop) asks for individuals that satisfy all clauses in the *body*. The (,) plays the role of logic conjunction (AND operation). Every clause in the *body* is deducted by its own rule. In the end, the deduction is backed by queries on the existing knowledge base. The isForStatement(Loop) is a more readable wrapper of the ontology query Loop is-a ForStatement, where the variable Loop binds to individuals if ontology triples (some-loop is-a ForStatement) exist in the knowledge base. Once the Loop is bound to some individual, clauses like hasForInit(Loop, InitExpr) search the knowledge base

Listing 4 Prolog specification of an OpenMP canonical loop (italic upper-case for variables, lower-case for properties).

```
1  % top level rule to find canonical loop
2  canonicalLoop(Loop) :-
3   isForStatement(Loop), !, %'!' prevents backtracking
4   hasForInit(Loop, InitExpr), %',' means logic AND
5   canonicalInit(InitExpr, LoopVar),
6   hasForTest(Loop, TestExpr),
7   canonicalTest(TestExpr, LoopVar),
8   hasForIncr(Loop, IncrExpr),
9   canonicalIncr(IncrExpr, LoopVar),
10  (
11   hasType(LoopVar, 'IntType'); %';' means logic OR
12   hasType(LoopVar, 'PointerType')
13  )
14  hasBody(Loop, ForBody),
15
16 % supportive rules to find canonical init-exp
17 canonicalInit(Init, LoopVar) :-
18  hasOperator(Init, AssignOperator), !,
19  hasLeftOperand(AssignOperator, VarRef),
20  referTo(VarRef, LoopVar),
21  hasRightOperand(AssignOperator, LB).
22
23 % rules with same heading: combined using logic OR
24 canonicalInit(Init, LoopVar) :-
25  hasVarDecl(Init, LoopVar),
26  hasInitializer(Init, Initializer),
27  hasValue(Initializer, LB),
28 % the rest is omitted ...
```

for (some-loop hasForInit some-init-expr) triples. Then canonicalInit(InitExpr, LoopVar) checks if the found initial expression individuals conform to the language specification. The query also returns the loop variable LoopVar if it can find it.

The analysis further checks whether the loop's init construct conforms to the specification. The first rule handles the var = lb style while the second rule deals with other styles. Different rules with the same head name form the logic disjunction (OR operation). The *cut* operator (i.e., the ! symbol) is used to prevent unwanted backtracking. It means that, as long as the first rule matches the form hasOperator(Init, AssignOperator), there's no need to check the second rule of the variable declaration form. Line 11 to 12 are the rules to do the type checking of loop variables. The (;) means logic disjunction (equivalent to two separated rules).

Experimental Results. The ontology in PATO successfully supports the analysis. We compare it with the imperative implementation in the ROSE compiler. The algorithm in ROSE traverses the AST tree of a program to find *for* statement nodes and check whether their sub-tree are in the canonical form. The code is written in C++ and is specific to the ROSE AST's internal data structures. The time complexity is $\mathcal{O}(n)$, where n is the number of AST nodes.

The code length of PATO-based analysis is about half of the imperative implementation in the ROSE compiler (190 versus 380 lines). We use the NAS Parallel Benchmarks (NPB) [8] for the measurement.

The results are shown in Figure 6. The frontend time corresponds to the parsing time of the *parser* in Figure 5. PATO uses the same frontend as the ROSE implementation. The

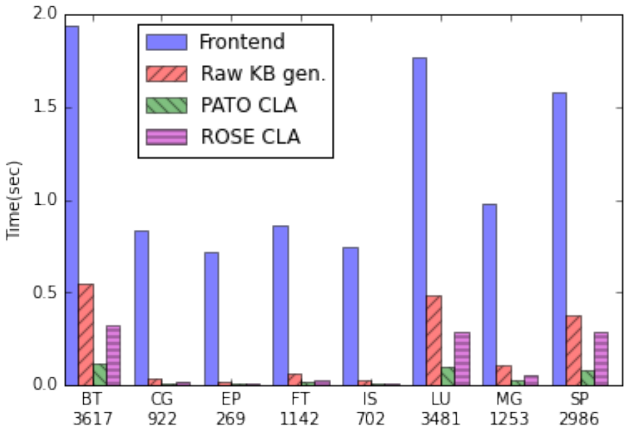


Figure 6 Time comparison of the canonical loop analysis. The X-axis shows the benchmark names and their numbers of source-code lines.

time of program knowledge generation (*KB gen*) represents the time taken by the ontology translator in the PATO framework. Finally, the other two bars are the time of the canonical loop analysis with PATO and ROSE respectively.

For the canonical loop analysis alone, we can see that the PATO declarative approach even beats the native implementation for most cases. It is slower only when the line number is small. One reason is that the ROSE's imperative implementation traverses the whole AST tree to find all for loops. For PATO's declarative implementation, the isForStatement relation information is stored explicitly and can be efficiently accessed by the hash-indexed keyword implementation of Prolog.

The ontology-based declarative analysis does have an obvious overhead: It needs an extra step to traverse the AST tree and build the initial knowledge base. However, as shown in the figure, the overhead is smaller than the frontend time. Besides, for the PATO system, the generated knowledge base can be reused for different program analyses (e.g., control flow graph analysis).

Extensibility. An appealing property of using ontology is the good extensibility. The previous discussion only covers canonical loops in C programs that use primitive loop variable types, such as pointer and integer types. OpenMP allows C++ canonical loops that use complex iterators as long as the iterator supports random access to the data elements. Examples of random access iterators include std::vector<T>::iterator and std::deque<T>::iterator. Also, programmers often define their own random access iterators. A conventional solution to extend an imperative CLA implementation is to store the known random access iterators, including the custom-defined ones, into a container for the compiler implementation to look up. The ad-hoc solution imposes barriers for exchanging the knowledge with other tools or developers. In PATO, adding such a support is simpler. One can easily add the concept RandomAccessIterator in the ontology and define it as is-a(Iterator)∧has(RandomAcess). The knowledge of is-a(Iterator) property can be gained by the knowledge builder while the knowledge has(RandomAcess) can be inserted into the knowledge base using the standard OWL API, either automatically by tools or manually by developers. The analysis rules can then reason about random access iterators. The resulting ontology can be then shared and reused by different analyses.

Table 1 Constraints in Andersen's pointer analysis.

Constraint type	Statement	Propagation rule
Base	$p = \&b$	$loc(b) \in pts(p)$
Simple	$p = q$	$pts(p) \supseteq pts(q)$
Complex	$p = *pp$	$\forall v \in pts(pp) \cdot pts(p) \supseteq pts(v)$
Complex	$*pp = q$	$\forall v \in pts(pp) \cdot pts(v) \supseteq pts(q)$

Listing 5 The Prolog rule to match the address-taken instruction.

```
1 matchAddressTaken(RHS) :- hasOperator(RHS, AddressOp),
2  hasOperand(RHS, LocRef), referTo(LocRef, Var).
```

5.2 Pointer Analysis

This part describes our experience in implementing Andersen's pointer analysis in PATO. Andersen's pointer analysis is a well-known inclusion based analysis [5]. The analysis result is typically represented as the points-to set pts(x) for each pointer variable x. It is flow-insensitive and does not distinguish different program execution points but computes what the pointers may refer to at any time of program execution.

The analysis is commonly regarded as solving a set constraint problem. It classifies assignments involving pointers into several kinds: taking the address of a stack variable or a heap allocated space, copying a pointer from one variable to another, and assignments through dereferences or references to a multilevel pointer. It defines a propagation rule (or called constraint) for each of the cases as illustrated in Table 1 (for C programs).

The analysis consists of two steps. The first step preprocesses the target program to produce a simplified intermediate representation, which keeps only the statements that involve pointer manipulations. Each kind of the assignments involving pointers is represented with a special notation. For example, $p = \&b$ is represented as stackLoc(p, b), $p = malloc(...)$ is represented as $heapLoc(p, ...)$, $p = *pp$ becomes load(p, pp), and $*pp = q$ turns into load(pp, q).

The second step propagates the points-to relations based on those constraints and solves the problem by computing graph transitive closures through an iterative worklist algorithm [5].

In our implementation on PATO, the preprocessing step is done on the program ontology representation through some simple pattern matching rules. For example, the address taken pattern ($p = \&b$) is matched by the following: The implementation breaks complex statements into simple ones by introducing temporary variables. For example, $*p = *q$ becomes $tmp = *q; *p = tmp$. Structures are analyzed in a field-sensitive manner, while an array is regarded a single data object as done in most previous studies [5, 27, 13].

For the analysis step, the propagation rules in Table 1 can be directly mapped into logic rules in Prolog as shown in Listing 6.

However, our experiments show that Prolog inferences based on these rules are not efficient. It evaluates the rules in a top-down manner with deep recursions and costly search through a large space. Inspired by previous work [13, 59], we instead implement the classic worklist algorithm in Prolog as illustrated as follows:

Line 11 in the listing means that as soon as Prolog finds out that there is no edge from A to P, it stops search and instead add A into the worklist and continue with next A' in

Listing 6 The Prolog rules for the points-to computation.

```
% p = &a; p = malloc()
pointsTo(P, Loc) :- stackLoc(P, Loc); heapLoc(P, Loc).
% p = q
pointsTo(P, X) :-  copy(P, Q), pointsTo(Q, X).
% p = *pp
pointsTo(P, X) :-  load(P, PP), pointsTo(PP, V), pointsTo(V, X).
% *pp = q
pointsTo(V, X) :-  pointsTo(PP, V), store(PP, Q), pointsTo(Q, X).
```

Listing 7 The worklist algorithm for pointer analysis in Prolog.

```
andersenPtr :-
 select(WorkList, V),
 propLoad(V); propStore(V);
 propFieldLoad(V); propFieldStore(V);
 propEdge(V),
 andersenPtr. % itratively execute the analysis

propLoad(V) :-
 load(P, V),
 pts(V, A),  % for each A in pts(V)
 (\+ edge(A, P) -> assertz(edge(A, P)), add(WorkList, A)).
% others are omitted
```

pts(V, A'). The use of $\rightarrow$ enables an early stop of useless searches, bringing large performance benefits.

The entire implementation takes less than 500 lines of code, about 300 lines of which are for the preprocessing step. When being applied to the NPB benchmarks, the analysis takes no more than 2 seconds on a program. We have also applied the analysis to three programs [2] with more pointer operations: bzip2 (7K lines of source code), gzip (8.6K lines), and oggenc (58K lines). The analysis times are 2.1 sec, 3 sec, and 36 sec respectively.

5.3 Facilitating Cooperations

In the final experiment, we try to examine the potential benefits of leveraging the standard representation of ontology to promote synergy between different compilers. Particularly, we use Liveness analysis as an example.

As a *may-type* data flow analysis, Liveness analysis is conservative—that is, if a variable belongs to the Liveout set of a basic block, it means that the compiler is uncertain whether the variable is dead at the end of the basic block. So, for two different Liveness analyses (both are sound and conservative), if a variable belongs to the result from one analysis but not that from the other, we can conclude that that variable is not Live at the end of the said basic block. In another word, the intersection of the results of the two Liveness analyses gives a more precise result than either of the two analyses.

Both LLVM Clang and ROSE have their own Liveness analysis developed before. By using the ontology converters we have developed for the two compilers, we convert the Liveness analysis results from them into our ontology. By writing several lines of Prolog

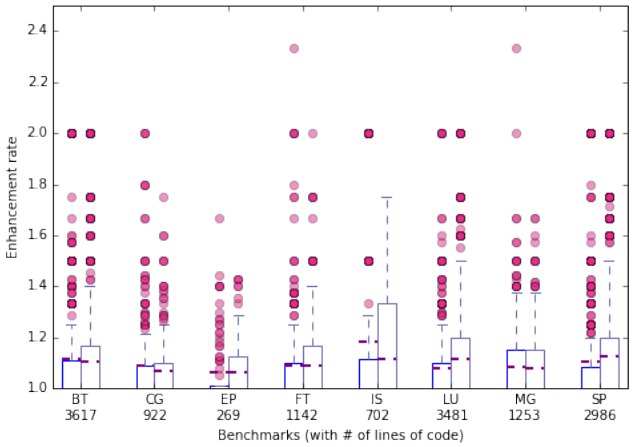


Figure 7 Enhancement of Liveness analysis by leveraging ontology to combine the results from the Liveness analysis in LLVM Clang and ROSE. (Dots show outliers). Left bars: the enhancement over LLVM Clang results; right bars: the enhancement over ROSE results.

code, the Prolog engine immediately extracts out the intersection of the sets of Live variables reported by the two compilers for each basic block of a given program.

We define a metric called *enhancement rate* to characterize the benefits of such a combination. Let A and B stand for two Liveness analyses and R_A and R_B be their Liveout set for a given basic block. The enhancement rate over A is defined as

$$enhancementRate(A) = \frac{|R_A|}{|R_A \cap R_B|}$$

The enhancement rate over B is defined similarly.

The results are shown in Figure 7. Box plots are used to show the distribution of the enhancement rates over all the basic blocks in the program. In the plot, the dots are outliers, and the intervals show the range of the lower 75 percentile of the enhancement in the observed result. The plot considers only the basic blocks whose Liveout sets are not empty in the result from at least one of the two compilers. The results indicate that the synergy improves the average precision of Liveness analysis by over 10% and 20% for LLVM Clang and ROSE respectively.

It is worth noting that the two compilers use different internal representations for programs and the Liveness analysis results. It is possible to write some special code to map their results to enable such a combination without using ontology. However, using the ontology designed in this work, the benefits come as simple side products of the ontology-based program analysis (by leveraging the converters and the standardized representation developed for many other program analyses). The productivity benefits would become even more prominent when many types of analyses cooperate across compilers.

5.4 Other Experience

For the interest of space, we briefly describe three other experiments.

One of them is an analysis to find out a program's array access patterns, including the access expressions to each array, lower and upper bounds of the loops surrounding an array reference, numbers of reads and writes to an array, and the ranges of its elements that are accessed. With PATO, each of the types of information can be easily extracted from the code

through just a few lines of Prolog statements. A previous imperative data access pattern analysis [14] has more than 2000 lines of source code. PATO, on the other hand, only needs 180 declarative rules to implement that analysis.

The second is to write code for control flow graph construction. It requires the examination of control flows of the whole program. On PATO, it is done through a set of simple rules based on the algorithm of inductive graph constructions [23]. We compare our PATO implementation with the implementations of ROSE and Clang [38], which also use the inductive construction algorithm but differ in the internal data structures and implementations. The PATO version uses only 400 lines of code, up to 8.75X shorter than the traditional compiler implementations (1200 and 3500 lines for ROSE and Clang respectively). The speed of the PATO analysis is similar to that of the Clang, and is 10-40% faster than that of the ROSE thanks to its efficient storage and query of the knowledge base.

Finally, we experiment with PATO for guiding data placement on Graphic Processing Units (GPU). A GPU has multiple types of memory with different performance characteristics. Prior studies have proposed some rule-based methods [34] and analytic model-based methods [14] for determine the placements of data that best suit the data access patterns of a GPU program and showed significant performance benefits. In this experiment, we build a data placement optimizer based on PATO, and find it much simpler to do than previous implementations. Because both the program knowledge and the hardware knowledge are represented in the standard triple format, they can be automatically linked into one single knowledge base, making automatic reasoning about data placement possible. It also simplifies the integration of heuristic rules and analytic models into a single analysis. Experiments on NVIDIA K20c GPU show that the PATO-based analysis gives similar placements as prior methods do [14]. Its analysis time is 5% longer, but it exhibits a better extensibility thanks to its ontology representation and declarative implementation. For example, in the previous work [14], to add the knowledge on some new GPUs memory features, one needs to extend a hardware specification language by adding some new constructs, and revise the code for parsing and performance modeling. In PATO, the needed changes are simpler: Through the Prolog GUI, users can easily add or remove a concept, and add queries on the new features into the analysis rules.

6 Related Work

Ontology has been used to build various knowledge bases in different domains, including Biology [6], Ambient Intelligence [19, 50, 51], Robotics [54], and others [43, 46, 49]. This work was enlightened by these studies, but concentrates on the special challenges facing program analysis.

In the software domain, ontology has been introduced, but mainly for software management and teaching of programming concepts, rather than program analysis. Specifically, Software ontology (SWO) [42] in the domain of software engineering focuses on the meta information of software (e.g., licenses, publishing processes, data formats). COPS [37] offers a sub-ontology for managing the knowledge related with image processing. Eden and others [21] have provide some theoretical discussions on the unique aspects in designing an ontology for programs, but without exploring the use of ontology for program analysis. There are several ontology designs for teaching some programming languages [52, 24]. This current work, to our best knowledge, is the first proposal on a systematic integration of ontology into program analysis.

There are some prior efforts trying to ease the difficulties in the development of program analysis. We discuss them in two aspects.

The first aspect is in the construction of a program analysis. Some prior studies have offered some interfaces for simplifying the construction of a program analysis. OpenAnalysis [3], for instance, introduces a set of analysis-specific interfaces (e.g., traverse all statements) as the building blocks for constructing a specific analysis. Other efforts in the same direction include GENOA [18] and StarTool [33]. These tools focus on imperative program analysis. There are some efforts that employ declarative program analysis to improve the productivity of program analysis development. JTransformer [1], for instance, is a tool integrated into Eclipes that allows the use of Prolog for analyzing and transforming Java source code. JunGL [57] introduces a scripting language for writing program analyses. It combines ML and Datalog. SemmleCode [58] is a tool that stores program-related data into a knowledge base, and allows the use of Datalog to write a program analysis that analyzes the program by querying the knowledge base. There are some other work [26, 13, 59] falling into the same category.

The second aspect is in the representation of a program. Some prior work has tried to develop a common software exchange format (SEF) for representing a program to facilitate the interoperation of software analysis and refactoring tools. Such a format needs both a schema (or called metamodel) that describes the objects and relationships, and a syntax that describes how model elements are to be stored and transmitted. The Dagstuhl Middle Metamodel (DMM) [40] is a representative of the former. DMM consists of a set of models that capture program elements and their relations. It follows some prior efforts such as Columbus [22] for C++ and the UML metamodel [11]. There are some other metamodels developed, such as Program Element Fact (PEF) developed in JTransformer [1], and DIMPLE [10]. Graphs are the most popular format for storing program elements in memory. TA [29] and TGraphs [20] are two examples, which are both based on typed graphs (i.e., directed graphs with attributes on both nodes and edges). GXL [30] was introduced as a generic way to use XML to represent such graphs. JunGL [57] uses some special graphs along with abstract syntax trees.

This current work shares some similarities with these prior studies, such as the use of logical programming to simplify the development of program analysis, and the creation of a common program representation. However, this work differs from the prior studies in several major aspects. First, it is the first work that points out the potential of ontology for the program analysis community to standardize the conceptualization for program analysis, and to promote the reuse and interoperations of analysis tools. It demonstrates the potential benefits through the Liveness analysis by two full-fledged existing compilers. Second, this work is the first that points out the benefits of ontology as a unified representation for not only program elements and relations but also knowledge from other sources (e.g., hardware knowledge) that are essential for program optimizations. It demonstrates the promise through GPU data placement optimizations. Third, unlike many of the prior studies that attempt to create a standalone tool for program analysis (e.g., JTransformer [1], Semmlecode [58]), this work aims to proposing an approach or a paradigm, which could potentially be employed by many program analysis tools and compilers, as demonstrated by the expeirments described in the previous section.

7 Conclusion

This work demonstrates the promise of ontology for overcoming the three major limitations of today's declarative program analysis. The five types of data analyses on PATO show that a single ontology is able to support multiple different program analyses efficiently. The

GPU data placement experiments indicate the promise of ontology for seamlessly linking knowledge from different sources, extending declarative program analysis with the capability to effectively guide program optimizations. The cooperative Liveness analysis demonstrates that with ontology-based program analysis, cooperations among different compilers or other program analysis tools become simple, and the synergy turns out to be quite beneficial. Overall, the study shows some promise of the integration of ontology into program analysis. Establishing this new way of program analysis, however, requires the development of an ontology for Program Analysis and some deep investigation of the opportunities that ontology may bring to program analysis and optimizations. We hope that this work will prompt further investigations by the community into this promising direction[2].

Acknowledgments. We thank the ECOOP'16 reviewers for the helpful comments.

References

1 The JTransformer project. `http://sewiki.iai.uni-bonn.de/research/jtransformer/`.

2 Large single compilation-unit C programs. `http://people.csail.mit.edu/smcc/projects/single-file-programs/`.

3 OpenAnalysis at Rice University. `http://www.hipersoft.rice.edu/openanalysis/`.

4 ROSE compiler infrastructure. `http://www.rosecompiler.org/`.

5 Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

6 Michael Ashburner, Catherine A Ball, Judith A Blake, David Botstein, Heather Butler, J Michael Cherry, Allan P Davis, Kara Dolinski, Selina S Dwight, Janan T Eppig, et al. Gene Ontology: Tool for the unification of biology. *Nature genetics*, 25(1):25–29, 2000.

7 Roberto Bagnara and Manuel Carro. Foreign language interfaces for Prolog: A terse survey. *ALP newsletter*, 15, 2002.

8 D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

9 Nick Bassiliades, Grigoris Antoniou, and Ioannis Vlahavas. A defeasible logic reasoner for the semantic web. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 2(1):1–41, 2006.

10 William C Benton and Charles N Fischer. Interactive, scalable, declarative program analysis: from prototype to implementation. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 13–24. ACM, 2007.

11 Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) Addison-Wesley Object Technology Series*. Addison-Wesley Professional, 2005.

12 Ivan Bratko. *Prolog (3rd Ed.): Programming for Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

13 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM. `doi:10.1145/1640089.1640108`.

[2] The source code of PATO and the analyses are accessible through the following links: https://github.com/yzhao30/PATO-ROSE, https://github.com/yzhao30/PATO-Pointer-Analysis.

14 Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. PORPLE: An extensible optimizer for portable data placement on GPU. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 88–100, Washington, DC, USA, 2014. IEEE Computer Society. doi:10.1109/MICRO.2014.20.

15 Michael Compton, Payam Barnaghi, Luis Bermudez, Raúl García-Castro, Oscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, Vincent Huang, Krzysztof Janowicz, W. David Kelsey, Danh Le Phuoc, Laurent Lefort, Myriam Leggieri, Holger Neuhaus, Andriy Nikolov, Kevin Page, Alexandre Passant, Amit Sheth, and Kerry Taylor. The SSN ontology of the W3C semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17:25–32, 2012. URL: http://www.w3.org/2005/Incubator/ssn/ssnx/ssn, doi:10.1016/j.websem.2012.05.003.

16 Leonardo Luiz Padovani Da Mata, Fernando Magno QuintãO Pereira, and Renato Ferreira. Automatic parallelization of canonical loops. *Sci. Comput. Program.*, 78(8):1193–1206, August 2013. doi:10.1016/j.scico.2012.09.006.

17 Steven Dawson, C. R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems—a case study. *SIGPLAN Not.*, 31(5):117–126, May 1996. doi:10.1145/249069.231399.

18 Premkumar T Devanbu. GENOA: A customizable language-and front-end independent code analyzer. In *Proceedings of the 14th international conference on Software engineering*, pages 307–317. ACM, 1992.

19 Ken Ducatel, Marc Bogdanowicz, Fabiana Scapolo, Jos Leijten, and Jean-Claude Burgelman. *Scenarios for ambient intelligence in 2010*. Office for official publications of the European Communities, 2001.

20 Jürgen Ebert, Volker Riediger, and Andreas Winter. Graph technology in reverse engineering–the TGraph approach. In *Proc. 10th Workshop Software Reengineering. GI Lecture Notes in Informatics*. Citeseer, 2008.

21 Amnon H Eden and Raymond Turner. Problems in the ontology of computer programs. *Applied Ontology*, 2(1):13–36, 2007.

22 Rudolf Ferenc, Árpd Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus-reverse engineering tool and schema for C++. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 172–181. IEEE, 2002.

23 Charles N. Fischer, Ronald K. Cytron, and Richard J. LeBlanc. *Crafting A Compiler*. Addison-Wesley Publishing Company, USA, 1st edition, 2009.

24 Gopinath Ganapathi, Ravi Lourdusamy, and Veeraraghavan Rajaram. Towards ontology development for teaching programming language. In *World Congress on Engineering*, 2011.

25 John H Gennari, Mark A Musen, Ray W Fergerson, William E Grosso, Monica Crubézy, Henrik Eriksson, Natalya F Noy, and Samson W Tu. The evolution of protégé: an environment for knowledge-based systems development. *International Journal of Human-computer studies*, 58(1):89–123, 2003.

26 Elnar Hajiyev, Mathieu Verbaere, and Oege De Moor. Codequest: Scalable source code queries with datalog. In *ECOOP 2006–Object-Oriented Programming*, pages 2–27. Springer, 2006.

27 Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation - PLDI '01*, pages 254–263, New York, New York, USA, 2001. ACM Press. doi:10.1145/378795.378855.

28 Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F Patel-Schneider, and Sebastian Rudolph. OWL 2 web ontology language primer. *W3C recommendation*, 27(1):123, 2009.

29 Richard C Holt. An introduction to TA: The tuple-attribute language. *University of Toronto, Toronto, Draft Mar*, 24, 1997.

30 Richard C Holt, Andreas Winter, and Andy Schürr. GXL: toward a standard exchange format. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 162–171. IEEE, 2000.

31 Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for OWL ontologies. *Semantic Web*, 2(1):11–21, 2011.

32 Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. *SIGSOFT Softw. Eng. Notes*, 20(4):104–115, October 1995. doi:10.1145/222132.222146.

33 Mark James and David Atkinson. STAR* TOOL- an environment and language for expert system implementation. *Jet Propulsion Laboratory Report NTR C*, 17536, 1988.

34 Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Trans. Parallel Distrib. Syst.*, 22(1):105–118, January 2011. doi:10.1109/TPDS.2010.107.

35 Mahmut Kandemir, J Ramanujam, and Alok Choudhary. Improving cache locality by a combination of loop and data transformations. *Computers, IEEE Transactions on*, 48(2):159–167, 1999.

36 Markus Krötzsch, Frantisek Simancik, and Ian Horrocks. A description logic primer. *arXiv preprint arXiv:1201.4089*, 2012.

37 Pascal Lando, Anne Lapujade, Gilles Kassel, and Frédéric Fürst. Towards a general ontology of computer programs. In *ICSOFT (PL/DPS/KE/MUSE)*, pages 163–170, 2007.

38 Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

39 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

40 Timothy C Lethbridge, Sander Tichelaar, and Erhard Plödereder. The dagstuhl middle metamodel: A schema for reverse engineering. *Electronic Notes in Theoretical Computer Science*, 94:7–18, 2004.

41 Chunhua Liao, DanielJ. Quinlan, JeremiahJ. Willcock, and Thomas Panas. Semantic-aware automatic parallelization of modern applications using high-level abstractions. *International Journal of Parallel Programming*, 38(5-6):361–378, 2010. doi:10.1007/s10766-010-0139-0.

42 James Malone, Andy Brown, Allyson L Lister, Jon Ison, Duncan Hull, Helen Parkinson, and Robert Stevens. The software ontology (SWO): A resource for reproducibility in biomedical data analysis, curation and digital preservation. *Journal of Biomedical Semantics*, 5(1):25, 2014.

43 Cynthia Matuszek, John Cabral, Michael J Witbrock, and John DeOliveira. An introduction to the syntax and content of Cyc. In *AAAI Spring Symposium: Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*, pages 44–49. Citeseer, 2006.

44 Deborah L McGuinness and Frank Van Harmelen. OWL web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.

45 Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2004.

46 Ian Niles and Adam Pease. Towards a standard upper ontology. In *Proceedings of the international conference on Formal Ontology in Information Systems-Volume 2001*, pages 2–9. ACM, 2001.

47 Natalya F Noy and Deborah L McGuinness. Ontology development 101: A guide to creating your first ontology, 2001.

48 OpenMP Architecture Review Board. OpenMP application program interface version 4.0, July 2013. URL: `http://www.openmp.org/mp-documents/spec30.pdf`.

49 Adam Pease, Ian Niles, and John Li. The suggested upper merged ontology: A large ontology for the semantic web and its applications. In *Working notes of the AAAI-2002 workshop on ontologies and the semantic web*, volume 28, 2002.

50 Davy Preuveneers, Jan Van den Bergh, Dennis Wagelaar, Andy Georges, Peter Rigole, Tim Clerckx, Yolande Berbers, Karin Coninx, Viviane Jonckers, and Koen De Bosschere. Towards an extensible context ontology for ambient intelligence. In *Ambient intelligence*, pages 148–159. Springer, 2004.

51 Natalia Díaz Rodríguez, Manuel P Cuéllar, Johan Lilius, and Miguel Delgado Calvo-Flores. A survey on ontologies for human behavior recognition. *ACM Computing Surveys (CSUR)*, 46(4):43, 2014.

52 Sergey Sosnovsky and Tatiana Gavrilova. Development of educational ontology for C-programming. 2006.

53 Steffen Staab and Rudi Studer. *Handbook on ontologies*. Springer Science & Business Media, 2013.

54 Moritz Tenorth and Michael Beetz. KnowRob—knowledge processing for autonomous personal robots. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 4261–4266. IEEE, 2009.

55 Dmitry Tsarkov and Ian Horrocks. FaCT++ description logic reasoner: system description. In *Automated reasoning*, pages 292–297. Springer, 2006.

56 Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Vol. I.* Computer Science Press, Inc., New York, NY, USA, 1988.

57 Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: A scripting language for refactoring. In *Proceedings of the 28th international conference on Software engineering*, pages 172–181. ACM, 2006.

58 Mathieu Verbaere, Elnar Hajiyev, and Oege De Moor. Improve software quality with SemmleCode: an Eclipse plugin for semantic code search. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 880–881. ACM, 2007.

59 John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 97–118, Berlin, Heidelberg, 2005. Springer-Verlag. `doi:10.1007/11575467_8`.

60 John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM. `doi:10.1145/996841.996859`.

61 Jan Wielemaker. SWI-Prolog Semantic Web Library 3.0. URL: `http://www.swi-prolog.org/pldoc/package/semweb.html`.

62 Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

63 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.